(12) UK Patent Application (19) GB (11) 2 253 500 (13) A

(43) Date of A publication 09.09.1992

(21) Application No 9117682.6

(22) Date of filing 16.08.1991

(30) Priority data
(31) 07571759 (32) 23.08.1990 (33) US

(71) Applicant
Data General Corporation

(Incorporated in the USA – Delaware)

4400 Computer Drive, Westboro,
Massachusetts 02171, United States of America

(72) Inventors
Paul U Conti
Jerome Lam
Eddie C Yuan

(74) Agent and/or Address for Service
Reddie & Grose
16 Theobalds Road, London, WC1X 8PL,
United Kingdom

(51) INT CL[5]
G06F 9/44

(52) UK CL (Edition K)
G4A AUD

(56) Documents cited
WO 90/04829 A2

(58) Field of search
UK CL (Edition K) G4A AUD
INT CL[5] G06F
Online databases: WPI, INSPEC.

(54) Object oriented-data bases

(57) A technique for creating and manipulating objects in a relational data base which responds to command call routines at a relational data base language level. A data base schema generation program is developed at a host program language level using abstract commands therefor. Such program language level commands are preprocessed to encode them into expanded command call routines at the program language level. Such program level command call routines are automatically converted into command call routines at the relational data base language level for use in creating and manipulating objects in the relational data base.

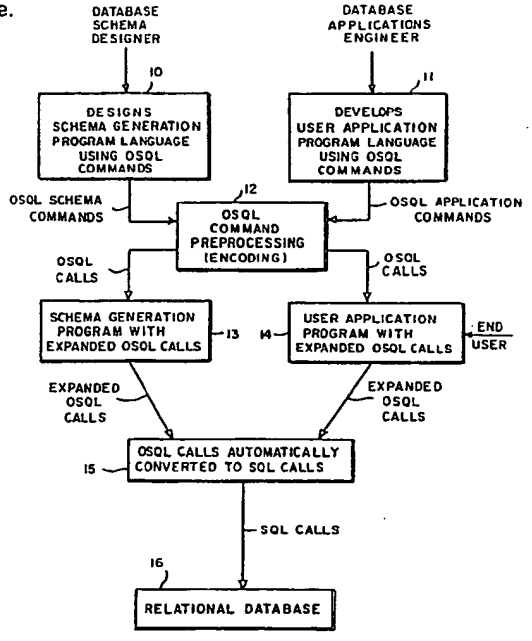

FIG. 1

At least one drawing originally filed was informal and the print reproduced here is taken from a late

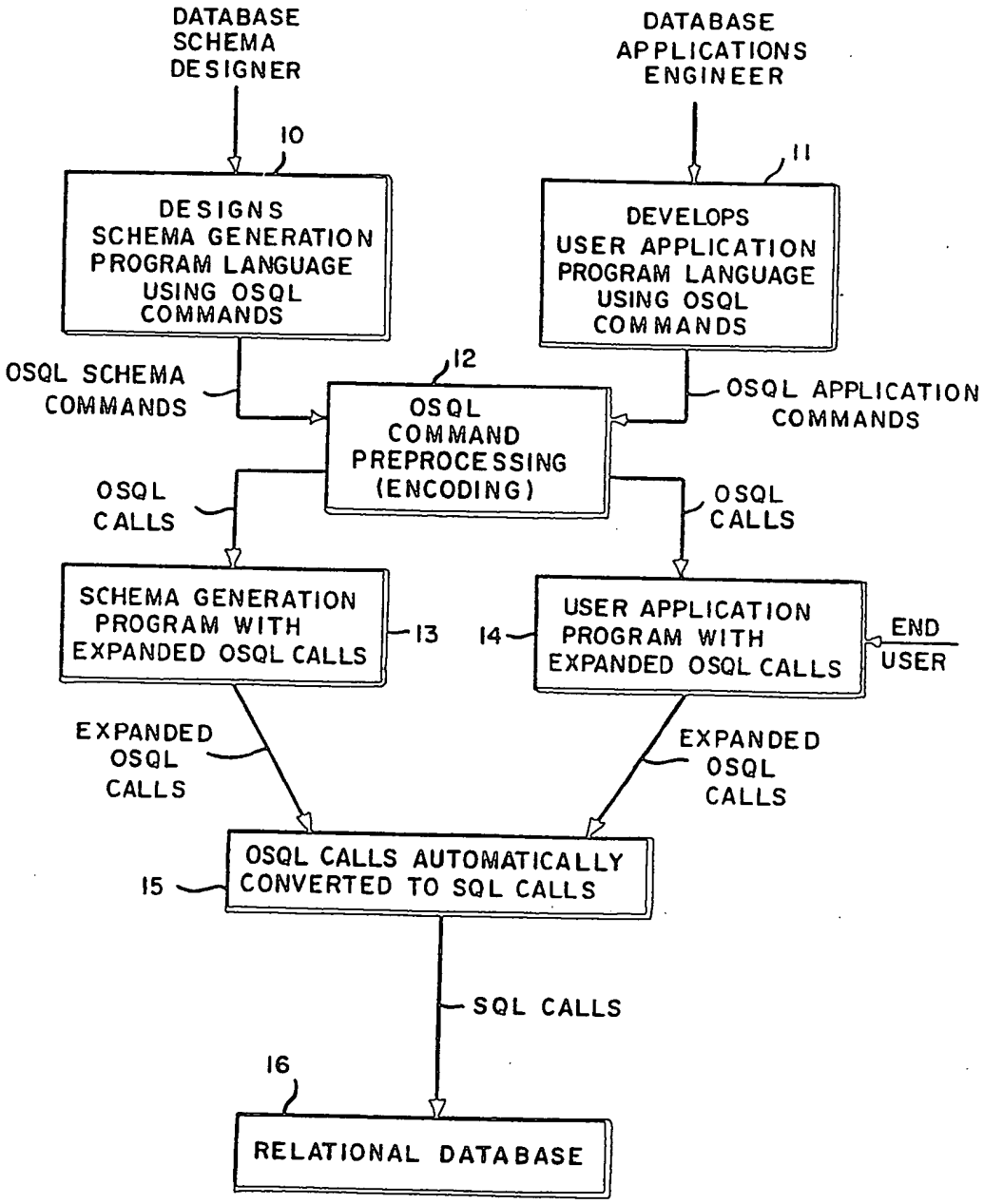

FIG. 1

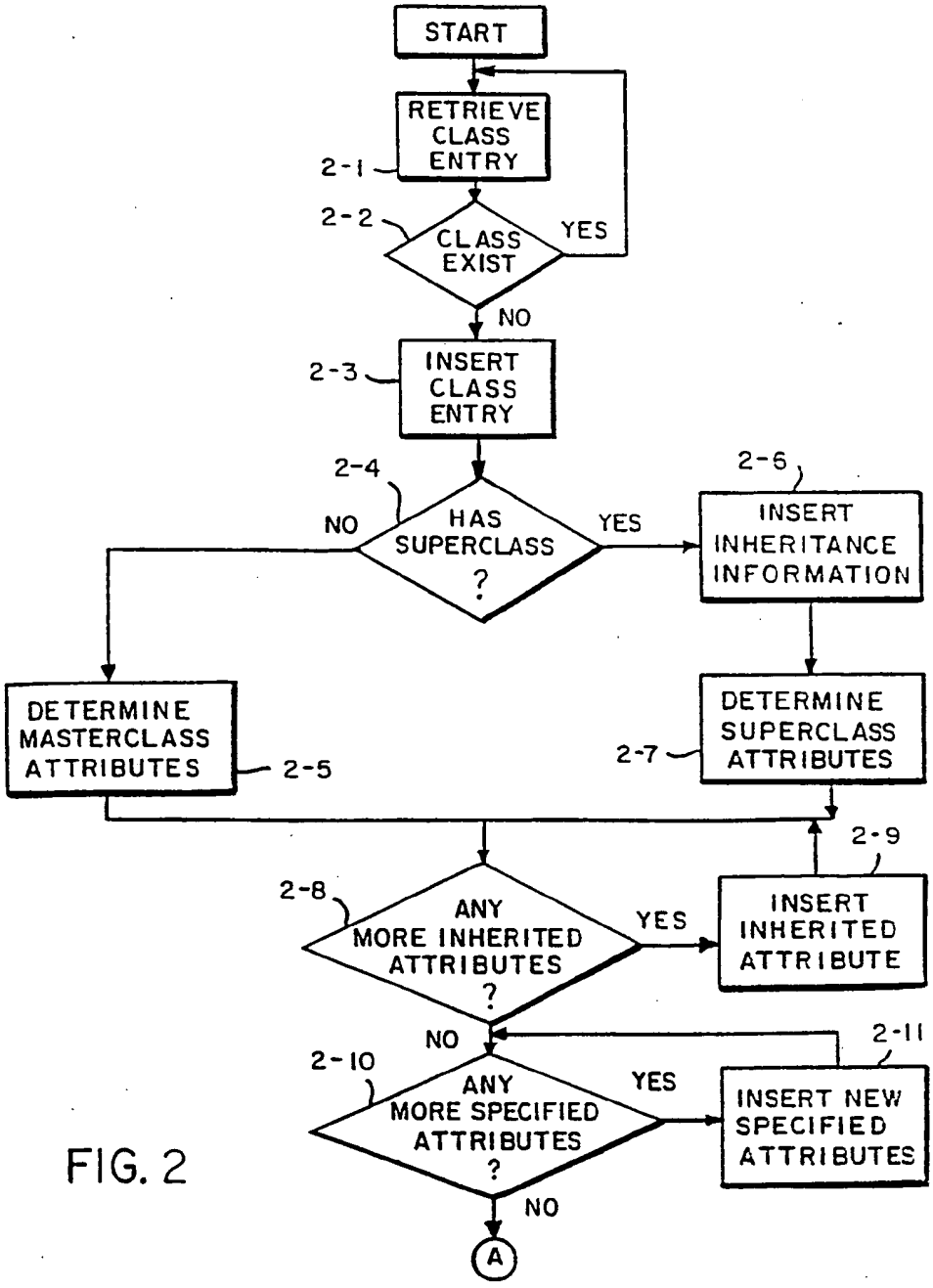

FIG. 2

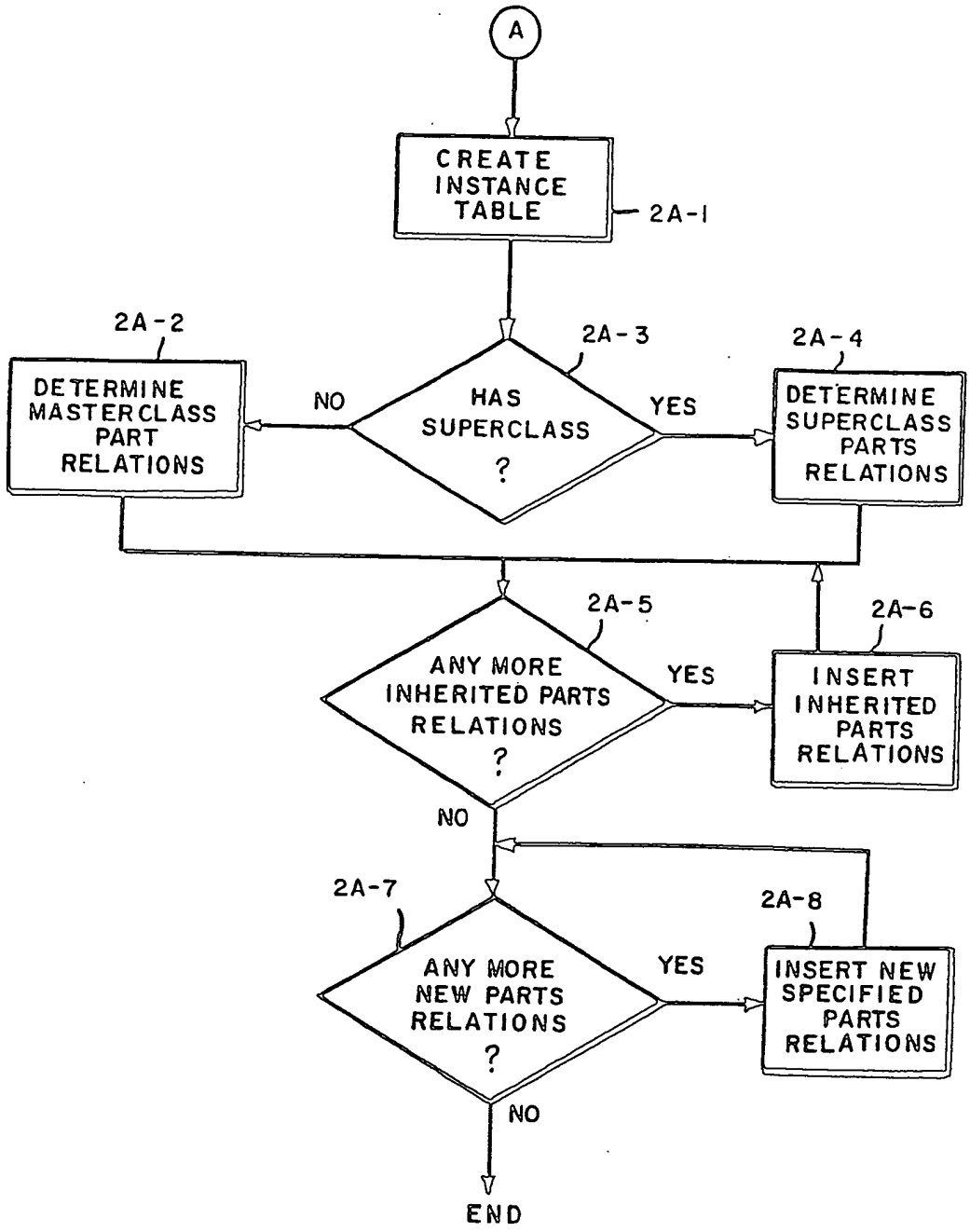

FIG. 2A

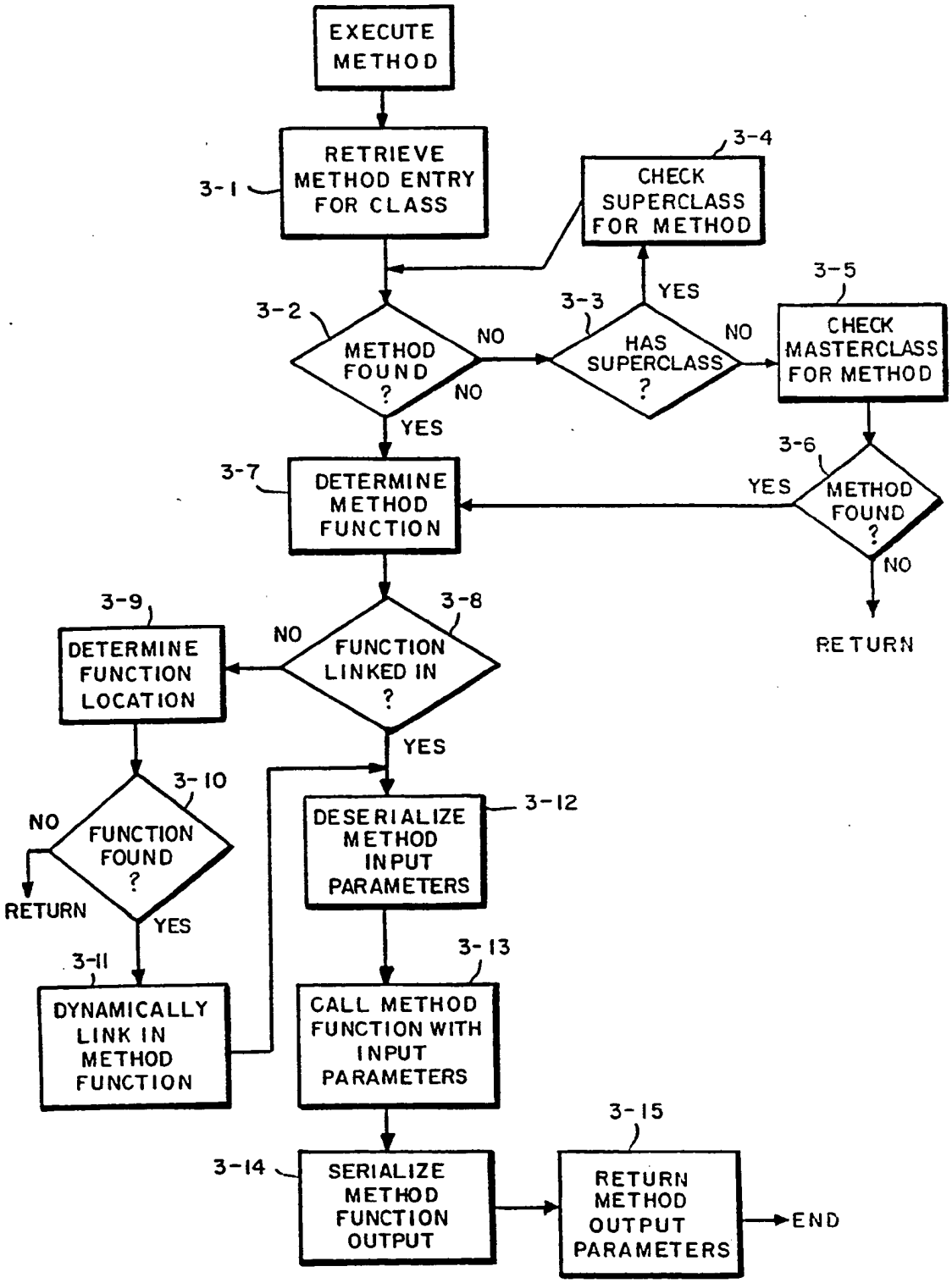

FIG.3

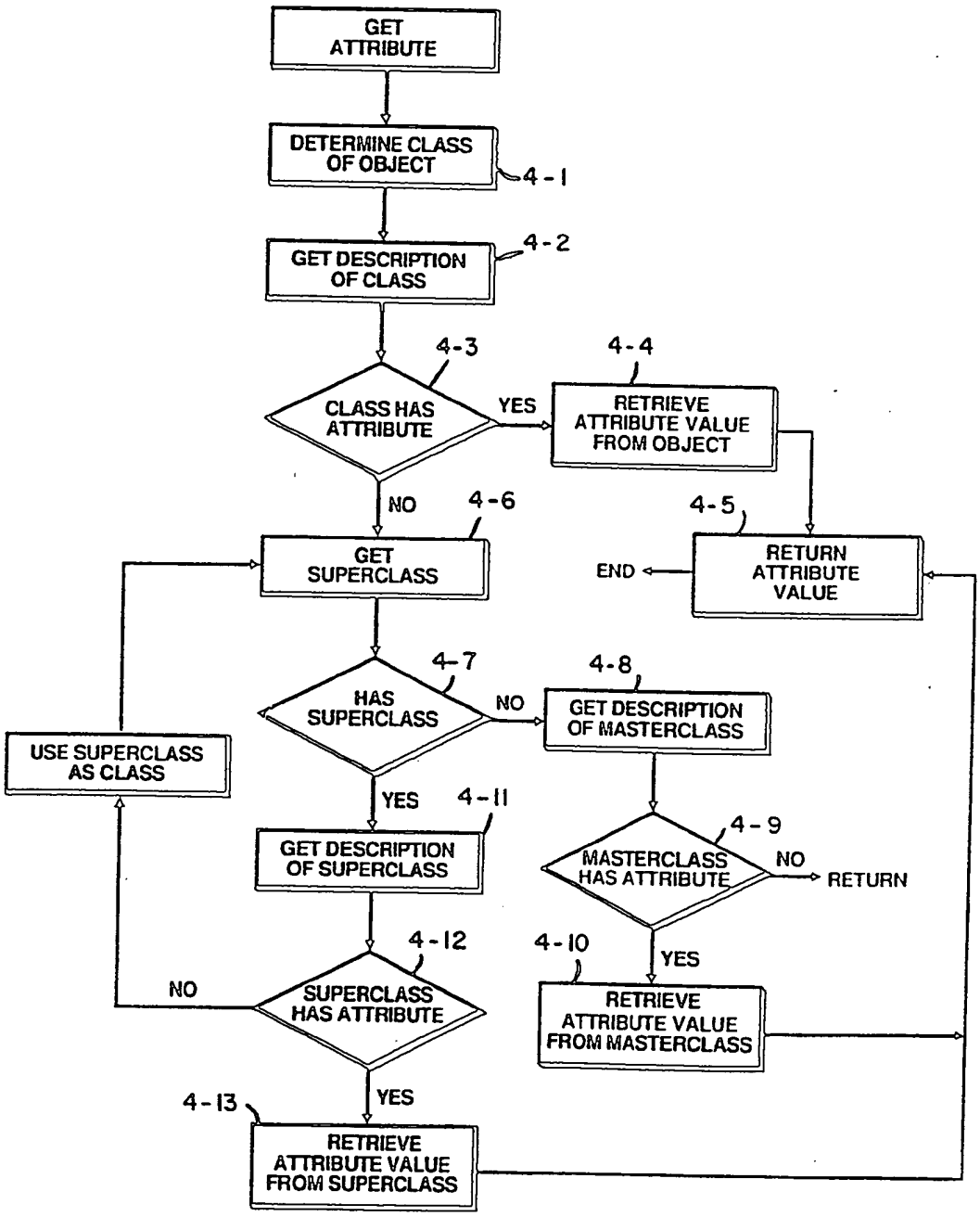

FIG. 4

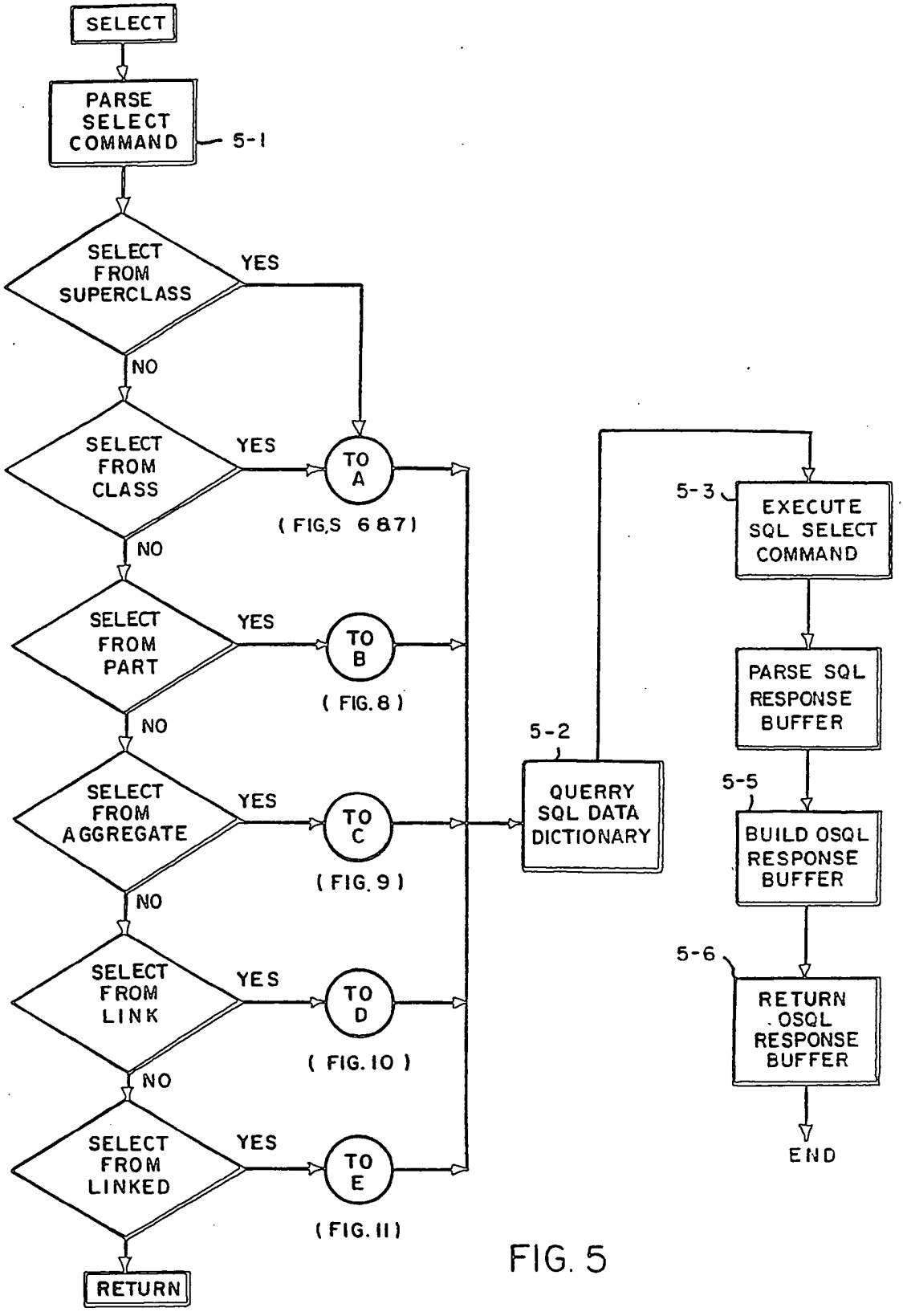

FIG. 5

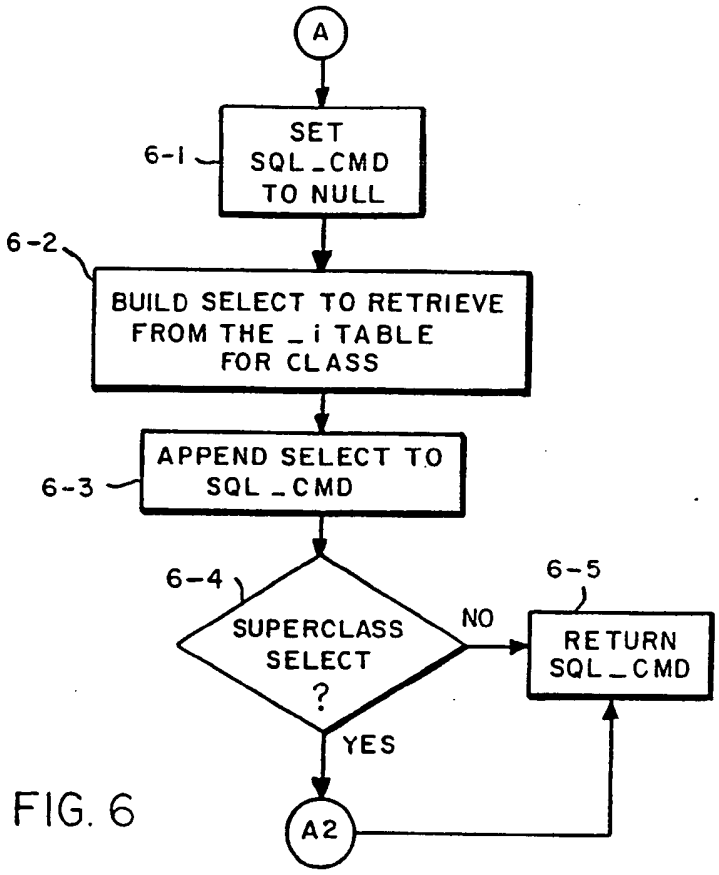

FIG. 6

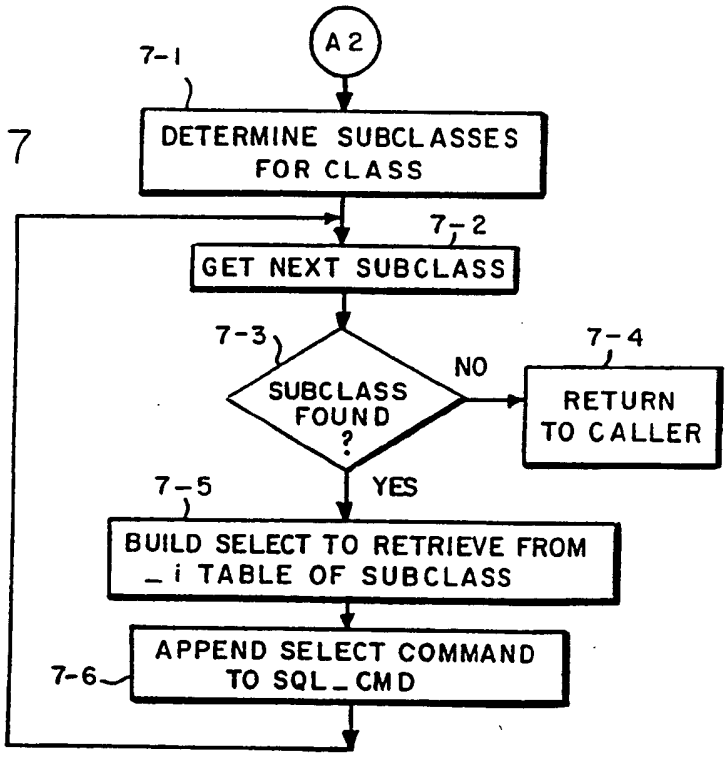

FIG. 7

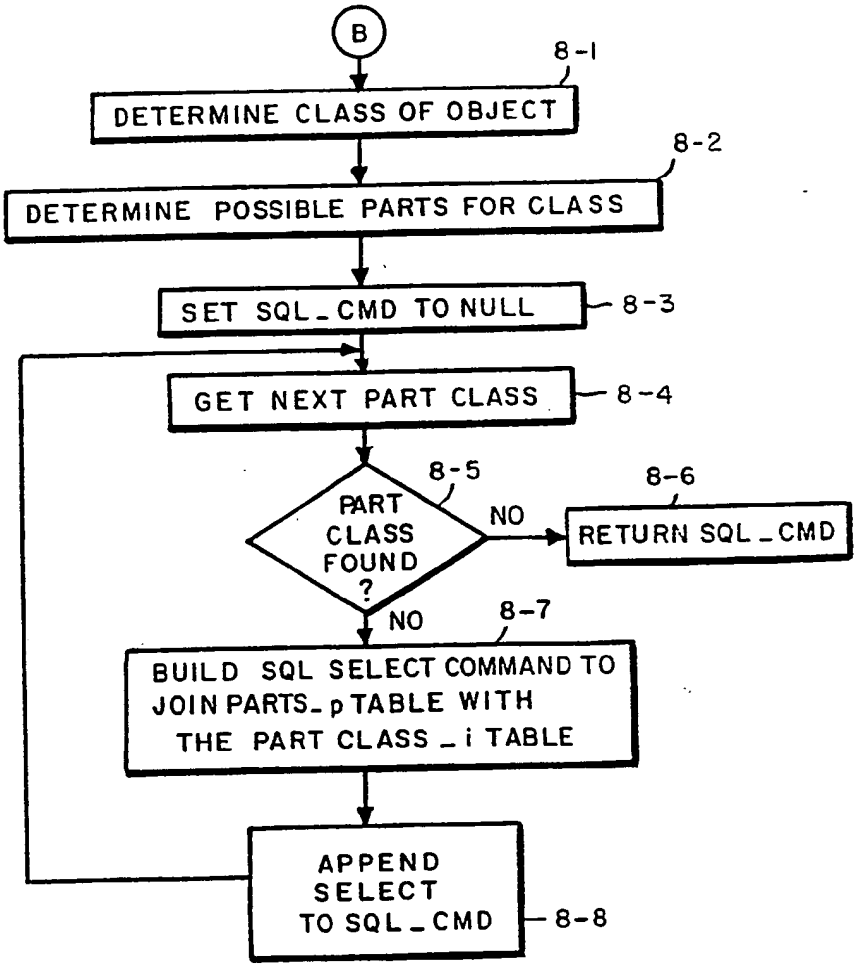

FIG. 8

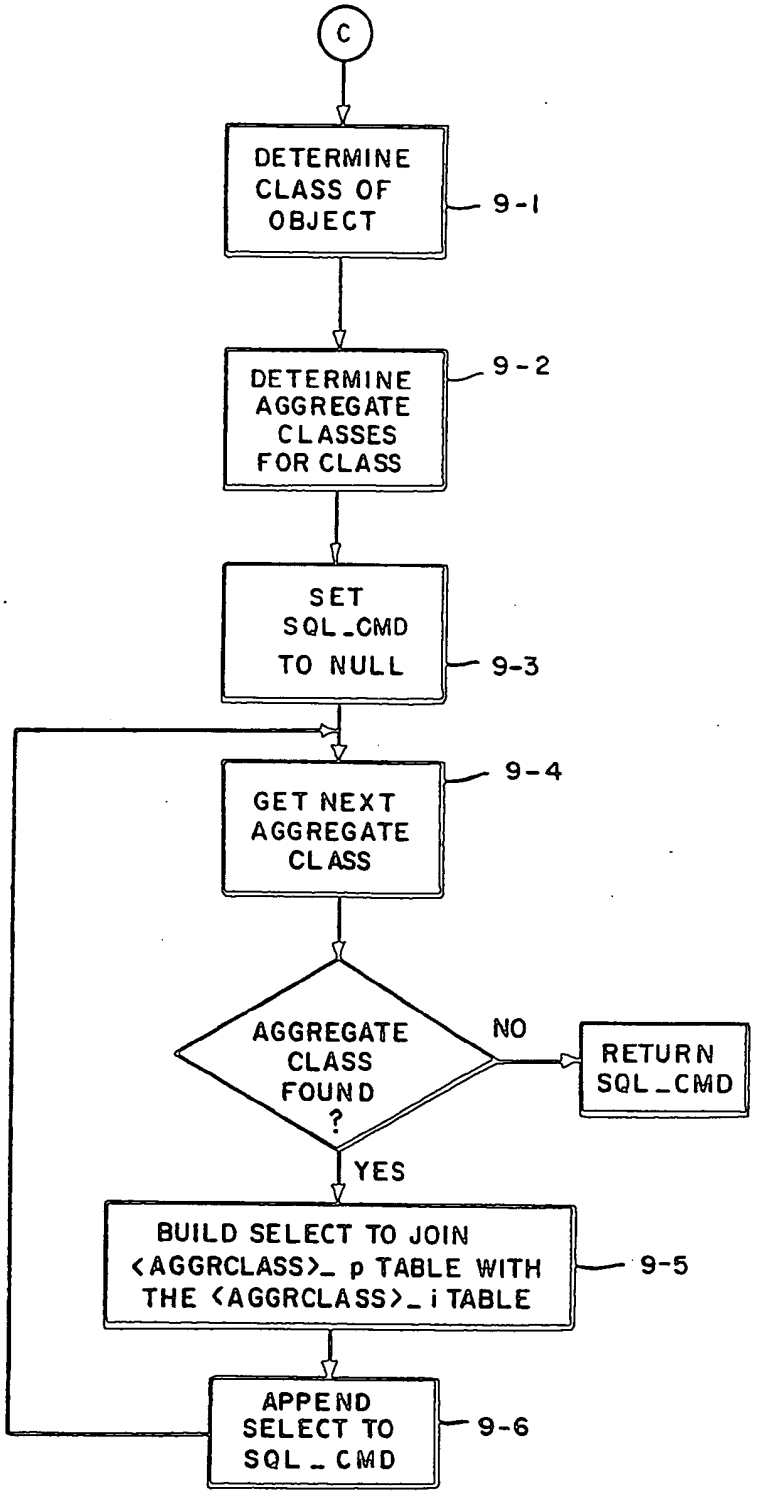

FIG. 9

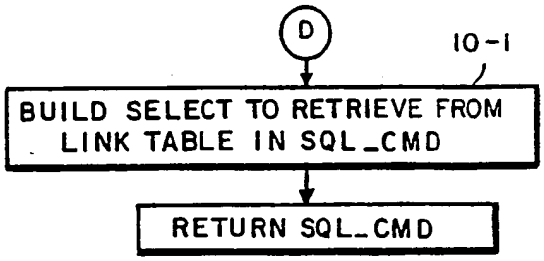

FIG. 10

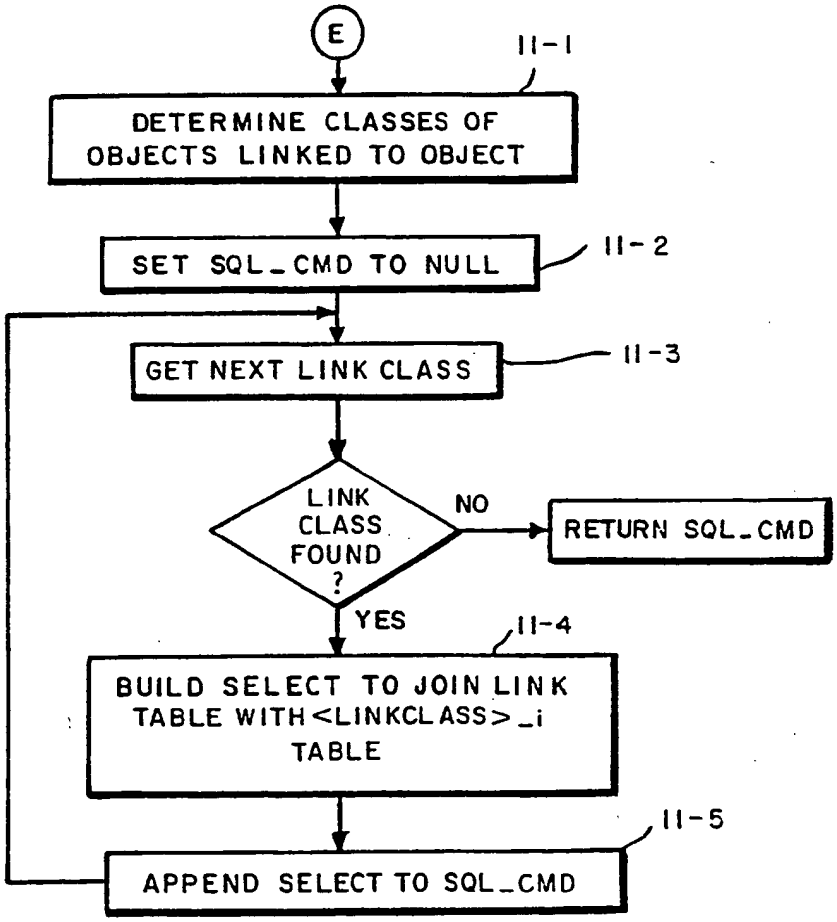

FIG. 11

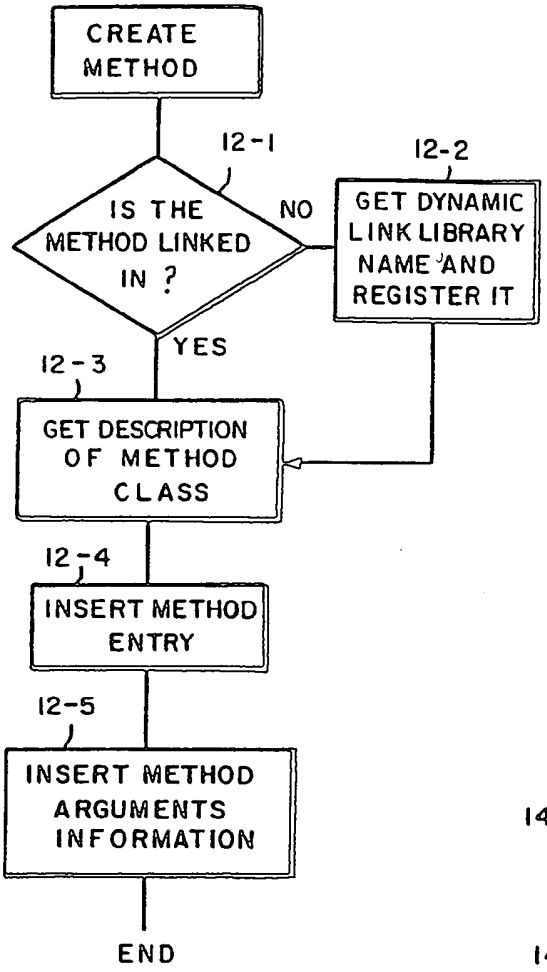

FIG. 12

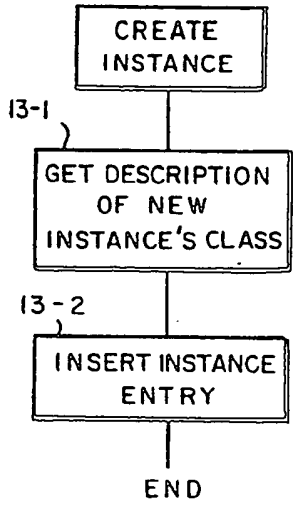

FIG. 13

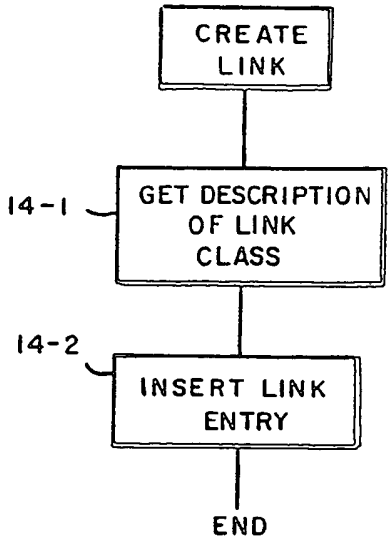

FIG. 14

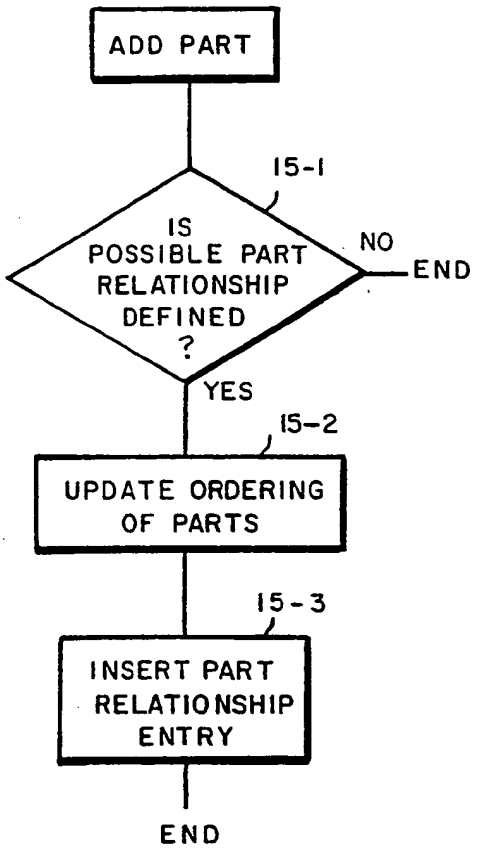

FIG. 15

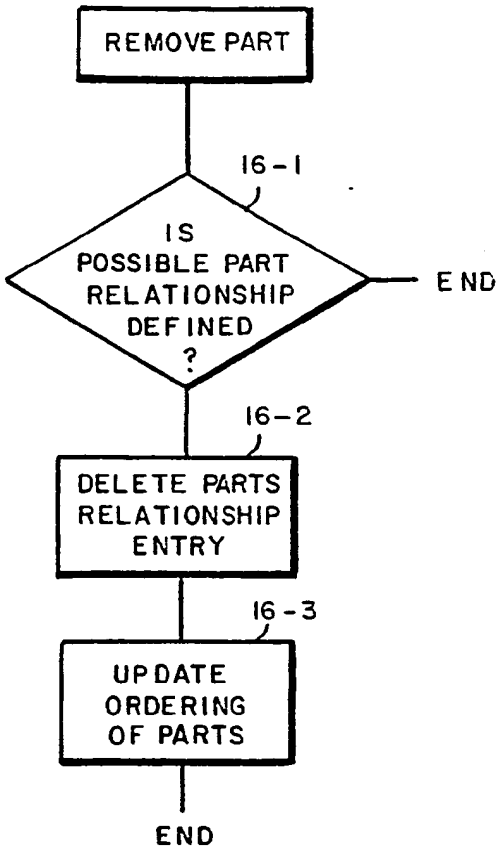

FIG. 16

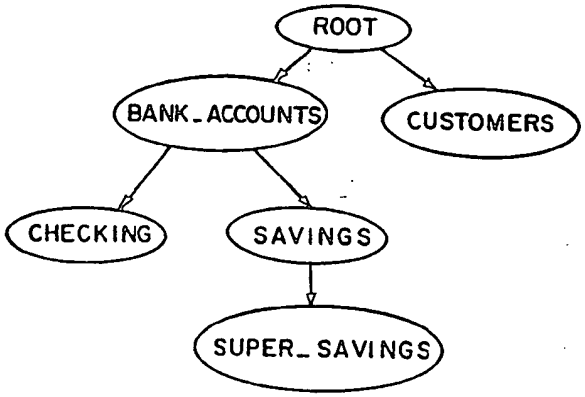

FIG. 17

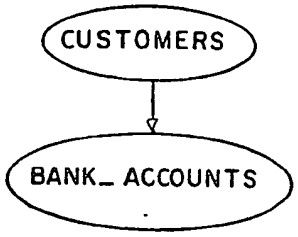

FIG. 18

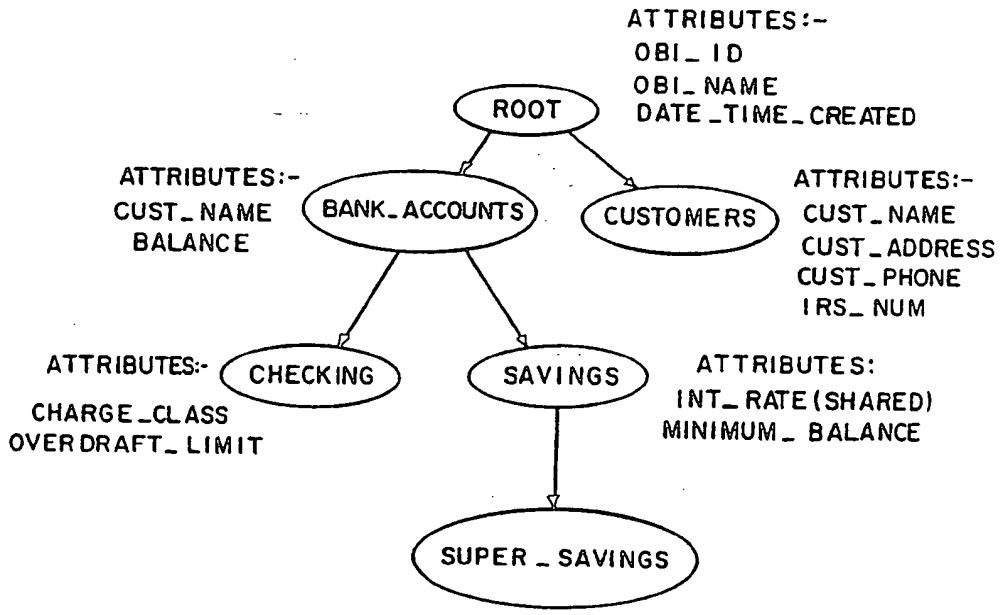

FIG. 19

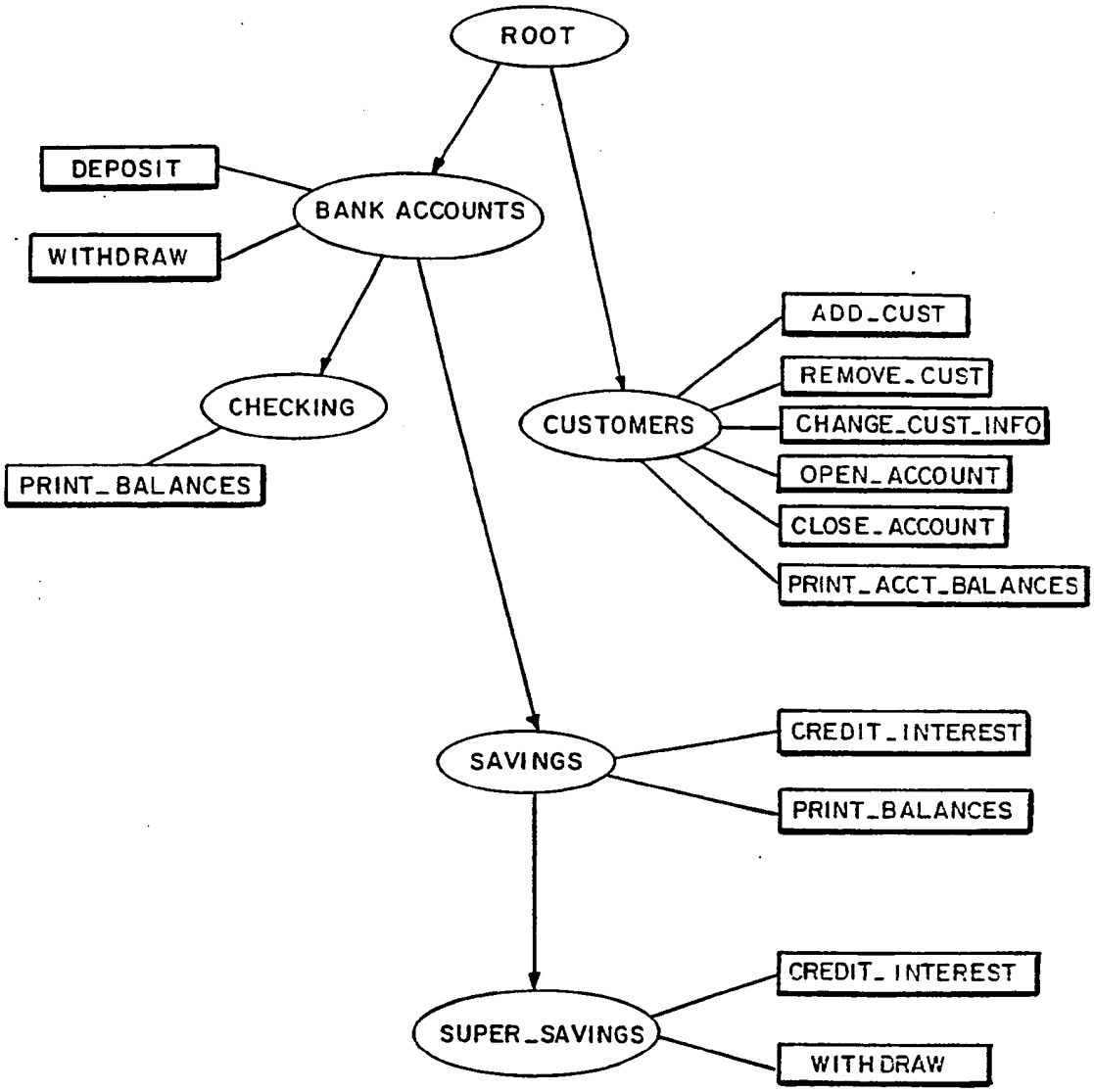

FIG.20

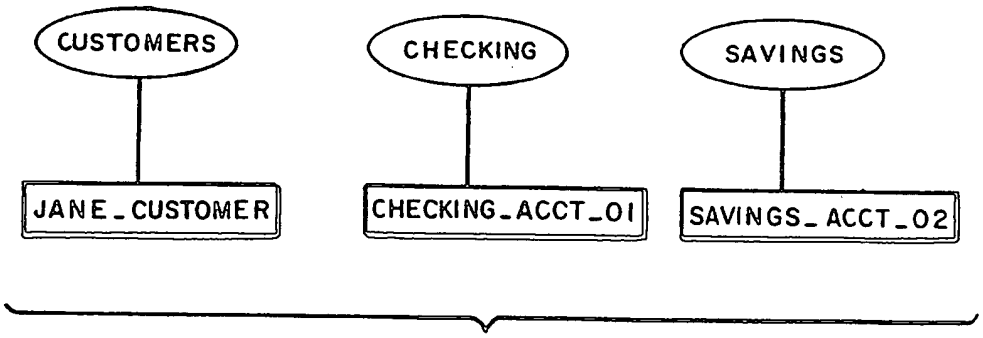

FIG. 21

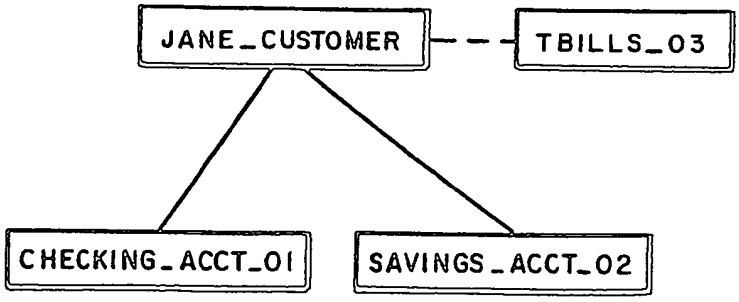

FIG. 22

OBJECT-ORIENTED DATA BASES

Introduction

This invention relates generally to the creation and use of object-oriented data bases for data processing systems and, more particularly, to the use of high level, object-oriented data base commands or call routines for use in creating and using a relational data base and to the automatic conversion of such high level object-oriented commands to lower level relational data base commands or call routines for such purpose.

Background of the Invention

As used herein the term "data base" is meant to refer to the information, or data, which is stored in a suitable storage device, e.g., a computer's internal memory or an external, or peripheral, memory device, such as a tape or a disk or the like.

Object-oriented data bases have been described and used by those in the art in which information concerning an object is placed in a custom designed data base, such information including information which defines the attributes of an object entity, information which defines methods, or procedures, for manipulating the object entity, and information which links, or relates, the object entity to other object entitites in the data base.

Relational data bases have also been described and used by those in the art. Conventionally, the information, or data, in a relational data base is normally stored in the form of records or tables and does not include information concerning procedures for manipulating such information. Accordingly, such data must be manipulated using a set of well known relational data base commands and call routines devised for use at a specified relational data base language level. One well known set of commands and call routines that have been used for such purpose, for example, is that designated as Structural Query Language, or SQL, commands which can be used both by a relational data base designer in designing a schema for creating the desired records and tables in a relational data base and by an applications software developer in preparing application programs for use in manipulating the information contained in the records and tables of the relational data base. End users make use of the data in the relational data base by using particular applications software programs which have been designed for such purposes.

In the traditional approach to the use of relational data bases, for each application for which the relational data base is to be used, the particular manipulations required for a particular application normally must be created from scratch for each such application by an applications software engineer.

In contrast, in object-oriented data bases, since the object entities thereon contain information concerning the manipulation thereof, such information does not need to be re-created from scratch each time an application software program is developed.

Since so many relational data bases are currently available for use by those in the art, it is desirable that a technique be devised for using them in a way that avoids the necessity for re-creating the procedures needed for manipulation of the data therein each time new application software is developed. Such a technique should provide a relatively easy way for data base schema designers, as well as application software designers, to achieve the advantages of the approach used in creating information contained as objects in object-oriented data bases, while at the time permitting such information to be stored for use in relational data bases which are already available to end users.

Brief Summary of the Invention

In accordance with the invention, a relational data base can be used for storing information which defines objects by developing a set of object-oriented data base commands and expanded command call routines which permit such objects to be defined at a program language level without requiring a knowledge of the relational data base commands and call routines of a specific relational data base in which such objects are to

be stored. Such object-oriented definitions include information concerning the attributes of the object which attributes characterize the object in a general way at such program language level, information concerning procedures which are used to manipulate the object, and information linking the object to other objects in the data base.

Once the objects are so defined at a program language level, a data base schema designer can communicate with a relational data base using a set of program language commands which can be expanded into program language data base command call routines, i.e., the program language commands are encoded into appropriately coded call routines at the program language level. The program language call routines are then automatically converted to a set of relational data base call routines, i.e., at the relational data base language level, for creating data, representing the defined objects, in the relational data base. The relational data base call routines are in effect invisible to a designer who needs only to have a knowledge of the program language command set and the encoding process for expanding the commands therein into the required coded call routines at the program language level but who does not need a knowledge of the particular commands and call routines required to use the relational data base at the relational data base language level.

Program language commands which are expanded into call routines for use with an SQL relational data base, for example, will sometimes be referred to herein as object SQL commands. As mentioned above, the use of object SQL commands at a program language level allows a data base schema designer to create objects in the relational data base using only a knowledge of the program language without having any knowledge of the SQL commands of the relational data base language.

Applications engineers can then further develop applications programs at the object SQL program language level, which programs can be used by an end-user for manipulating the objects created in the relational data base, also without requiring any knowledge of the SQL relational data base language commands on the part of the applications engineer or the end user. The commands and call routines at the program language level are automatically and invisibly converted to call routines at the relational data base language level for such purpose.

Thus, in an SQL data base context, an object oriented data base system of the invention automatically transforms or converts object SQL call routines (i.e., expanded commands) into relational data base SQL call routines in accordance with controlled programs which support the object-oriented paradigm of the invention, as described in more detail below. By such transformation the SQL call routines, when executed by the

system, then permit the creation and manipulation of object information in the relational data base.

Description of the Invention

The invention can be described in more detail with the help of the accompanying drawings wherein

FIG. 1 is a block diagram depicting providing an overall operational view of the invention;

FIG. 2-16 show flow charts of command call routines forming a representation basis set thereof for use in the invention; and

FIG. 17-22 show block diagrams depicting the relationships used in a specific example of an application of the invention as related to a banking system.

In describing the invention it is helpful to review below the concepts which are used in the creation and use of object-oriented data bases, as well as to provide specific examples thereof which will assist in better understanding such concepts.

Basic Object-Oriented Data Base Concepts

An object-oriented data base (ODB) is based on the idea of object-orientation, wherein the object-orientation approach

models an instance object as an instance of a class that is encapsulated with its attributes, procedures, and links to other instance objects. Listed below in accordance with the approach of the invention to object-oriented data bases are the following main concepts that can be used to define object-orientation. A specific example is then later discussed to assist in better understanding such concepts and how they can be used in the invention.

CLASS	A class is a grouping of objects with similar attributes and/or behavior.
INSTANCE	An instance is an object that is a member of a Class.
METHODS OR PROCEDURES	A method or procedure is a set of operations that is applicable to the objects.
HIERARCHY	Classes are organized in hierarchies so that a class can itself consist of other classes. This concept allows complex objects to be built from existing classes and avoids duplication of work that would occur if all objects had to be defined from scratch for each new application program.

INHERITANCE With such a hierarachical organization of classes, a class and therefore the instance of a class may inherit attributes or procedures from other classes.

LINKS A link is a way of associating objects.

ENCAPSULATION One of the goals for an object-oriented data base is to encapsulate the data base object, that is, to provide all of the necessary information and operations for manipulation of the object as a part of the object, access to the object being available only through defined methods. Encapsulation is attained through the use of class, instance, methods, hierarchy, inheritance, and links.

Object Oriented Database Schema

In accordance with the invention, an object oriented database schema is divided into seven kinds of relationships for the above described four kinds of object data base objects, i.e., classes, instances, methods, and links. Such relationships and objects are defined below and examples are provided to assist in understanding such relationships.

IS-TYPE-OF	This relationship specifies that a given class is a subclass of another class (i.e., the other class can be designated as a superclass). The subclass can inherit procedures and attributes from the superclass.
CAN-HAVE-PARTS	This relationship specifies that a given class can have other classes as parts thereof (i.e., the other classes can be designated as subclasses). Such relationship specifies that the class is an aggregate class.
IS-INSTANCE-OF	This relationship specifies that a given object is an instance of a class.
HAS-METHOD	This relationship specifies that a class object has a method object.
HAS-ATTRIBUTE	This relationship specifies that a class or instance object has an attribute.
HAS-PARTS	This relationship specifies that a given instance object has other instances as parts thereof, i.e., it specifies that the instance is an aggregate instance.

HAS-LINK	This relationship specifies that a given instance object has a linked object which associates two instance objects.

In designing an object-oriented data base a database schema designer must understand the nature and complexity of the objects that will be stored in the object oriented database and must define the characteristics and behavior of objects by grouping the objects into classes. The characteristics of objects must be defined by specifying the attributes and superclasses for the class. The behavior of objects must be defined by specifying the relationships with other classes and specifying the methods which control the actions of objects. This body of information is called the database schema and is defined by using suitable schema commands in a schema generation program which is written by a schema designer.

Further a database application software developer must understand the database schema, as defined by a database schema designer, as well as how the objects will be used by an end user. The software developer writes program language applications software with embedded program commands to produce a user application program which uses appropriately selected commands so as to provide the result desired by an end user

thereof, i.e., the program language that contains the embedded program commands allows an end user to manipulate objects in a database.

In accordance with the invention, if an object-oriented approach is to be used for creating in effect an object-oriented data base in a relational data base context, the technique therefor is arranged so that the objects can be created by a schema designer at a program language level so that the objects can be placed in the relational data base without any need to know the specific relational data base program commands. Thus, in order to use a relational data base requiring relational data base commands and call routines, e.g., SQL program commands and call routines in an SQL relational data base, the program level commands utilize higher level, object-oriented commands which when used with an SQL data base, for example, can be referred to herein as object SQL commands.

The technique includes a preprocessing operation wherein the program devised at the program language level has embedded therein such program level object SQL commands which are preprocessed to produce expanded program language commands, i.e., object SQL command call routines which are in effect encoded object SQL commands. Such program language object SQL call routines are then automatically converted to SQL call

routines at the relational data base language level which are used, for example, to create objects in the relational database.

Applications software can also be developed at the program language level using such object SQL commands which are likewise preprocessed to produce program language object SQL call routines for automatic conversion to SQL call routines at the relational data base program level. An end user can then use such an applications program at the program language level to retrieve and manipulate objects in the relational database.

Thus, in accordance with the technique of the invention, the encoded schema command call routines or the encoded applications command call routines in program language are automatically converted into relational database command call routines in a controlled way so as to support the object oriented paradigm. Such conversion procedure consults the schema information when executing database commands, responds to object oriented database call routines at the program language level (object SQL call routines) and provides command call routines at the lower data base program level (SQL call routines) for creating, retrieving or manipulating objects, i.e., data stored as objects in the underlying relational database. The data base can be any commercially available relational database such as an SQL data base, which can be used as the underlying mechanism to store object data base records and tables as further discussed below.

FIG. 1 shows a block diagram representing an overall view in an operational sense of an object oriented database (ODB) system of the invention which is to be used, for example, with an SQL relational data base. As can be seen therein, a data base schema designer designs a schema generation program using program language object SQL commands (block 10) or a data base applications engineer develops a user applications program also using such program language object SQL commands (block 11). Such object SQL commands in either case are preprocessed (encoded) to expand such commands so as to produce object SQL command call routines for use either for a schema generation program (block 13) or for a user application program (block 14). Such expanded command call routines are automatically converted to SQL call routines (block 15), the SQL call routines then being used appropriately to create objects in an SQL relational data base in accordance with the data base schema program or to retrieve or manipulate the objects as created in an SQL relational data base in accordance with the application program (block 16). Such a system is in effect a multi-layered system that supports a set of object SQL program language commands which are transformed or converted, in a manner invisible to a programmer, from the object oriented data base paradigm to relational data base language call routines at the relational data base level. The latter process (block 15) of FIG. 1 is an operational layer which is sometimes referred to

as an Object SQL engine, the function of which is to provide an effective conversion process, as explained in more detail below.

The object SQL engine effectively provides the knowledge and means for converting or transforming the object oriented paradigm to the relational database model. As shown in Chart A below, the object SQL engine suports three sets or categories of object oriented command groups, identified as Schema Commands, Manipulation Commands, and Retrieval Commands.

The set of schema commands allows a database schema designer to define the characteristics and behavior of objects. The specific data information defining such characteristics and behavior is stored in the relational database. The database schema designer utilizes his knowledge of what defines an object, how those objects behave, and the relationships of the objects with other objects in implementing such commands.

The set of manipulation commands allows both a database schema designer and a database applications engineer to manipulate objects, while the set of retrieval commands allows both a database schema designer and a database applications engineer to retrieve objects from the database.

The representative sets of the above types of commands, shown in Chart A can be considered as effective basic, or core, sets thereof for use in the invention. It should be noted, however, that other additional commands may also be devised by those in the art within each set for any particular application. The particular sets set forth can then be thought of as relatively basic sets effectively required for most or all systems in which the invention will be found useful.

CHART A

Schema Commands	Manipulation Commands	Retrieval Commands
Create Class	Create Instance	Get Attribute
Create Method	Add Part	Select From Class/ Superclass
	Remove Part	Select From Subclass
	Create Link	Select From Part
	Execute Method	Select From Aggregate
		Select from Link
		Select From Linked

In order to support the object oriented paradigm, the system stores information in a set of ODB tables (sometimes referred to as "meta-tables" and "user-defined" tables) which are present and available in the underlying relational database.

Meta-tables are used for the control or management of objects and contain information which a system consults frequently to carry out the operation of the system. User-defined tables are created by a user to contain information concerning user-defined objects in a specific application, for example, the specified information therein concerning particular objects created by or for a specified user.

Meta tables are internal to the system and contain information (referred to a meta information) that the system consults frequently to carry out its operation, which tables are not seen by ordinary users of the system. A list of exemplary meta tables useful in a system of the invention is set forth below:

root
objectattributes
objectid
parts_m
class_i
class_m
inheritance_m
link_i
method_i
methodattributes

Appendix A provides descriptions of such meta tables.

User-defined tables are also invisible to ordinary users of the system. Specific examples of user-defined tables, as set forth below, are described in Appendix B for use in a particular example of a banking system, as discussed in more detail below.

customers_i
bank_accounts_i
checking_i
savings_i
super_savings_i
tbills_i

Flow charts showing the steps required for executing some of the above representative core commands in the command sets of Chart A are shown in FIGS. 2-16, each of said commands being discussed below. Such commands are implemented in a program language.

In the conversion of such program commands to relational data base commands, coded program modules are created so that each step of the steps shown in a flow chart, which are used in executing the command illustrated by the flow chart, utilizes the required relational data base command, or commands, for implementing such step in the flow charts are discussed below.

Description of Create Class Command (FIGS. 2 and 2A)

This command is used to add a new class in the CLASS table of the relational data base. At the start of the Create Class Command, the CLASS table in the relational data base is queried

before creating a new class entry therein for a newly specified class in order to determine if the desired new class already exists in the table. If the proposed new class does exist, a return is made. If such class does not exist, an entry for the new class is inserted into the CLASS table.

A check is also made to determine if any superclass/subclass entry has already been specified for the new class. If not, a superclass/subclass entry for any superclass specified is inserted into the INHERITANCE table. The CLASS ATTRIBUTE table is queried to retrieve the attributes for the superclass specified for the new class since these attributes will be inherited by the new class. If no attributes for a superclass exist, the CLASS ATTRIBUTE table is further queried to retrieve the attributes for the masterclass which is a superclass of all classes by default and these attributes will be inherited by the new class. A check is made to see if there are any additional inherited attributes and, if so the next inherited attribute is inserted into the CLASS ATTRIBUTE table.

A check is made to see if there are any new attributes specified for the new class and, if so, the next new specified attribute is inserted into the CLASS ATTRIBUTE table. An __i Instance Table is created for the new class. This table has a column for each inherited and specified attribute of the new class. A check is made to see if any superclasses were

specified for the new class. The POSSIBLE PARTS table is queried to retrieve the possible parts for the superclasses specified for the new class, since these possible parts relationships will be inherited by the new class. If there are no such possible parts, the POSSIBLE PARTS table is queried to retrieve the possible parts for the materclass which is a superclass of all classes by default. These possible parts relationships will be inherited by the new class.

A check is made to see if there are any more inherited possible part relationships. If so, the next inherited possible parts relationship is inserted into the POSSIBLE PARTS table and a check is made to see if there are any new possible part relationships specified for the new class. If so, the next new possible part relationship is inserted. A __p Parts Table is created, i.e., a map table that will be used for joining the parent and parts instance table. At this stage the class has been duly created and the Create Class Command ends.

Description of Execute Method Command (FIG. 3)

The Execute Method Command starts and a query is made of the METHOD table to determine if the method to be executed can be found therein. If such method is not found, a check is made to see if the class has a superclass from which the method can be inherited. If the class has a superclass, the METHOD table is queried to obtain the method entry for the superclass. If the

class does not have a superclass, the METHOD table is queried to obtain the method entry for the masterclass which is a superclass to all classes by default.

If the masterclass does not have a method, the method does not exist for the specified class, either directly or through inheritance, so a return is made. If the masterclass has a method, either directly or through inheritance, specified function associated with the method is determined.

Once the method is found (either for a class, a superclass or a masterclass, a check is made to determine if the function is linked in the ODB process. If function is not so linked, a specified library is used to obtain the function. For this purpose a check is made to see if the specified function library can be found. If the specified function library cannot be found, a return is made. If the specified function library is found, a dynamic linker is used to link the method function into the ODB process. Since the execute method was called with the method input parameters serialized in a buffer, the parameters are deserialized. The method function is invoked and the input parameters are passed in.

After a return from the method function, the output parameters are serialized into the response buffer and the

serialized output parameters are returned to the caller. At this point the execute method command is ended.

Description Of Get Attribute Command FIG. 4)

The Get Attribute Command will retrieve a specified attribute value for the object specified by an object id. This value will be retrieved either directly from the object or through inheritance if the object does not have the attribute.

The OBJECT ID table is queried to determine the class of the specified object and to obtain a description of the class of the specified object. The CLASS ATTRIBUTE table is queried to determine if the class has the desired attribute. If the class has the desired attribute, the __i instance table for the class is queried and the attribute value for the specified object is retrieved.

If the specified class does not have the desired attribute, the INHERITANCE table is queried to obtain the superclass for the class and a check is made to see if the class has a superclass. If the class has a superclass, the CLASS ATTRIBUTE table is queried to determine if the superclass has the desired attribute. If the superclass does not have the desired attribute the superclass is used as the class and a determination is made to see if it in turn has a superclass. If

a superclass has the attribute, the attribute value stored in the CLASS ATTRIBUTE table which is a shared value, is used.

If the class does not have a superclass the CLASS ATTRIBUTE table is queried to determine if the masterclass has the desired attribute (the masterclass is a superclass to all classes by default). If the object does not have the specified attribute either directly or through inheritance from a superclass or the masterclass, a return is made. If the masterclass has the attribute, the attribute value stored in the CLASS ATTRIBUTE table, which is a shared value, is used. The attribute value is returned to caller and the Get Attribute Command ends.

Description of Select Command (FIG. 5)

A Select Command is used to select attributes from one of a plurality of attributes tables. At the start of Select Command, the select command is parsed to determine the type of select (i.e., the type of attributes table to be selected), the attributes selected, and the selection criteria. Thus, the type of select is checked to see if it is a select of an attributes table for a superclass, a class, a part, an aggregation, a link or a linked. If the type of select is unknown, a return is made.

An SQL select command is then constructed, as described with reference to FIGS. 6-12, to perform either a select from class

or a select from superclass, a select from part, a select from aggregation, a select from link or a select from linked.

Following a construction of the specified SQL select command, the SQL data dictionary is queried to determine the amount of memory required to store the attributes returned by the SQL select command that was so constructed. The SQL select command is executed using the memory to store the attributes from a buffer. The SQL response buffer returned by the select which is stored in the memory is parsed and an object SQL response buffer is constructed using the attributes returned by the SQL select command. The object SQL response buffer is returned to the caller and the select command ends.

Description of Build Select From Class/Superclass (FIG. 6)

This command starts a module that builds (constructs) the SQL select command that will provide the results for the object SQL select from class/superclass command. This command will retrieve the specified attributes from the specified class/superclass.

The sql__cmd buffer for storing the SQL command is initialized by setting it to null. The SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will act in accordance with the specified class's __i instance table. The

constructed SQL select command is appended to the sql__cmd buffer and the type of select is checked to see if it is from a superclass. The constructed SQL command is retrieved.

With respect to this command the select is a select from superclass which treats the specified class as a superclass and therefore selects information from the specified class and all subclasses of that class. It basically treats the class and all of its subclasses as a single class. It calls the A2 module that recursively constructs the SQL select command for the subclasses.

Description of Build Select From Subclasses (FIG. 7)

This command starts a module that builds the SQL select command for the subclasses of the specified class and then recursively calls itself to handle the subclasses of the subclasses.

The INHERITANCE table is queried to get the subclasses for the class that is passed in. The next subclass is obtained from the list of subclasses for the passed in class and a check is made to see if there is a subclass. If there are no more subclasses for the passed in class, a return is made to the point where this module was called.

If a subclass is found the SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will act in accordance with the subclass's __i instance table. The constructed SQL select command is appended to the sql__cmd buffer. Since this constructed SQL select command and the previously constructed SQL select commands must act as a single command, the UNION keyword is inserted between the selects so that the relational database will merge the selects.

The subclass is now parsed to the A2 module to recursively build the SQ1 select for the subclass's subclasses.

Description of Build Select From Part (FIG. 8)

This command starts a module that builds the SQL select command that will provide the results for the object SQL select from part command. This command will retrieve the specified attributes from the part objects of the object specified by an object id.

The OBJECT ID table is queried to get the class of the specified object. The POSSIBLE PARTS table is queried to determine what classes of objects the specified object can have as parts. The sql__cmd buffer for storing the SQL command is initialized by setting it to null.

The next class that is a possible part for the specified object is obtained and a check is made to see if there is a possible parts class. If there is no more possible parts class, the sql__cmd returns to the point where the module was called.

If a possible parts class is found, the SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will join the class's __p parts table and the part class's __i instance table to get the parts objects for the specified object. The constructed SQL select command is appended to the sql__cmd buffer. Since this constructed SQL select command and any previously constructed SQL select commands must act as a single command, the UNION keyword is inserted between the selects so that the relational database will merge the selects.

Description Of Build Select From Aggregate (FIG. 9)

This command starts a module that builds the SQL select command that will provide the results for the object SQL select from aggregate command. This command will retrieve the specified attributes from the aggregate (parent) objects of the object specified by an object id.

The OBJECT ID table is queried to get the class of the specified object. The POSSIBLE PARTS table is queried to determine what classes of objects can have the specified object

as a part. The sql__cmd buffer for storing the SQL command is initialized by setting it to null.

The next class that can be an aggregate for the specified object is obtained and a check is made to see if there is an aggregate class. If there are no more aggregate classes, the sql__cmd returns to the point where the module was called.

If an aggregate class is found, the SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will join the aggregate class's __p parts table and the aggregate class's __i instance table to get the object that has the specified object as a part. The constructed SQL select command is appended to the sql__cmd buffer. Since this constructed SQL select command and any previously constructed SQL select commands must act as a single command, the UNION keyword is inserted between the selects so that the relational database will merge the selects.

Description of Build Select From Link (FIG. 10)

This command starts a module that builds the SQL select command that will provide the results for the object SQL select from link command. This command will retrieve the specified attributes from the LINK table. The SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will act in

accordance with the LINK table. The constructed SQL select command returns to the point where this module was called.

Description Of Build Select From Linked (FIG. 11)

This command starts a module that builds the SQL select command that will provide the results for the object SQL select from linked command. This command will retrieve the specified attributes from the objects linked to the object specified by an object id.

The LINK table is queried to determine the classes of objects that are linked to the specified object. The sql__cmd buffer for storing the SQL command is initialized by setting it to null. The next class of objects linked to the specified object is obtained and a check is made to see if there is a linked class.

If there are no more linked classes, the sql__cmd returns to the point where the module was called. If a linked class is found, the SQL select command is built to retrieve the specified selected attributes using the specified selection criteria. This select will join the LINK table and the linked class's __i instance table to get the objects that are linked to the specified object. The constructed SQL select command is appended to the sql__cmd buffer. Since this constructed SQL select command and any previously constructed SQL select command

must act as a single command, the UNION keyword is inserted between the selects so that the relational database will merge the selects.

Description of Create Method (FIG. 12)

Creating a method object is similar to creating other kinds of objects in the system, a difference being that it is a kind of "meta" object information as to which is kept by the system. For example, there are a Method - i table and a Method attributes table, the former storing method objects and the latter storing method arguments.

An inquiry is made to determine if the method is already linked in. If not, no further activity is required. If the method is linked in, a description of the method meta class is obtained and a method entry is inserted. Method argument information is also inserted and the program ends.

Description of Create Instance (FIG. 13)

In creating an instance a description of the new instance's class is obtained and the instance entry is inserted, as shown.

Description of Create Link (FIG. 14)

In a similar manner in order to create a link, a description of the link meta class is obtained and the link entry is inserted as shown.

Description of Add Part (FIG. 15)

A determination is made as to whether a possible part relationship is already defined. If not, the program ends. If such relationship has been defined an update ordering of the parts occur and the part relationship entry is inserted, as shown.

Description of Remove Part (FIG. 16)

In an effective inverse operation that adding a part, as above, a determination is made as to whether a possible part relationship has already been defined. If not, the program ends. If such relationship has been defined, the part relationship entry is delected and an update ordering of the parts occurs.

The process as described in the flow charts of FIGS. 2 - 16 shows how the core object SQL commands involved are translated into SQL statements. Certain numbered blocks in the flowcharts (e.g., blocks 2-1, 2-2, 2-3, etc. in FIG. 2 etc.) contain the required SQL translations, the following specific SQL statements being shown and discussed below as corresponding to such numbered blocks of the flow charts.

In understanding the explanation below, it should be understood that the symbol is used to represent a value that is provided to SQL statements at runtime because of their

dynamic nature. A short explanation of the value being expected is enclosed in the symbol . Wherever a occurs in SQL statements below, it should be noted that the use of such a symbol is not a normal SQL syntax but rather is a convenient way of representing dynamic values for the explanation below.

Further, some SQL statements below utilize the symbol "?", which can be recognized a valid SQL syntax to mean that the values involved will be determined at runtime.

In the figures, the following blocks contain the SQL statements required to translate the object SQL step into the SQL call routines required in each case.

2-1.

```
SELECT object_id
    FROM Class_i
    WHERE object_name = < name of the class being created >;
```

2-2.

By examining the return code of the SQL statement in Step 2-1, it can be determined whether or not the class already exists.

2-3.

A list of attributes is obtained from the Class_i table (see Appendix A). Information about these attributes is used to

insert an entry into Class_i. Following is the SQL statement that retrieves attributes information:

```
SELECT attribute_id, attribute_name, attribute_type,
        inherited, table_number, flags
    FROM ObjectAttributes
    WHERE object_id = < A constant representing
                        Class meta object >;
```

The above SELECT statement will be PREPARE'd and a cursor will be DECLARE'd and OPEN'ed to hold all the attributes information about the Class_i table.

An SQL statement is built dynamically at runtime to INSERT an entry into the Class_i table. Following is the statement:

```
INSERT INTO Class_i ( object_id, object_type, class_name,
                      time_created, created_by,
                      time_modified, modified_by, ... )
        VALUES ( < determined at runtime > );
```

As shown above, the INSERT statement will be created dynamically and eventually executed. After an entry is successfully INSERT'ed into the Class_i table for the new class, a new class object is created.

2-4.

This Step tests if any superclass was specified in the OSQL CREATE CLASS statement. Number of superclasses specified is an argument passed to the create class function.

2-5.

```
SELECT attribute_id, attribute_name, attribute_type,
       inherited, table_number, flags
    FROM ObjectAttributes
    WHERE object_id = < An integer constant representing the
          object id of RootClass >;
```

2-6.

```
INSERT INTO Inheritance_m ( obj1, obj2, priority )
   VALUES ( < class_id >, < superclass_id >, < priority > );
```

2-7.

```
SELECT attribute_id, attribute_name, attribute_type,
       inherited, table_number, flags
    FROM ObjectAttributes
    WHERE object_id = < superclass's object id >;
```

2-8.

This Step tests is whether other superclasses are specified. If so, the attributes of each superclass should be inserted into ObjectAttributes table as shown in Step 2-9.

2-9.

```
INSERT INTO ObjectAttributes ( object_id, attribute_id,
                               attribute_name,
                               lowerc_attr_name,
                               attribute_type, inherited,
                               table_number, flags )
    VALUES ( ?, ?, ?, ?, ?, ?, ?, ? )
```

The above statement is PREPARE'd and appropriate values will be provided for "?"'s at runtime. The statement is executed in a loop, each attribute of the new class will have an entry INSERT'ed into the ObjectAttributes table.

2-10.

This Step tests if there are any user-specified attributes in the OSQL CREATE CLASS statement. If so, they should be inserted into ObjectAttributes table as shown in Step 2-11.

2-11.

Step 2-11 is identical to Step 2-9.

2A-1.

A CREATE TABLE statement is used to create a new table to hold new objects of this class. At this point, all the attributes have been inserted into the ObjectAttributes table. A SELECT FROM ObjectAttributes table similar to that in Step 2-5 will be executed to get the attributes information into a cursor. The cursor is then FETCH'ed to 'build' a CREATE TABLE statement dynamically at runtime. This CREATE TABLE will eventually be executed and a new table will be created for this new class.

2A-2.

```
SELECT obj2, has_array, minimum, maximum
    FROM Class_m
    WHERE obj1 = < RootClass's object id >;
```

obj2 above means the object id of all the part classes of the RootClass or MasterClass. This statement is PREPARE'd and a cursor is OPEN'ed on it. For each part class id in the cursor a part relationship will be created between the new class, as the owner, and the part class as the owned class.

2A-3.

This Step tests if any superclass was specified in the OSQL CREATE CLASS statement. Number of superclasses specified is an argument passed to the CREATE CLASS function.

2A-4.

```
SELECT obj2, has_array, minimum, maximum
    FROM Class_m
    WHERE obj1 = < object id of the superclass
                   of the new class >;
```

obj2 above means the object id of all the part classes of the of the new class. This statement is PREPARE'd and a cursor is OPEN'ed on it. For each part class id in the cursor a part relationship will be created between the new class, as the owner, and the part class as the owned class.

2A-5.

In addition to explicitly defined CAN-HAVE-PART relationships ODB creates CAN-HAVE-PART relationship between class objects when appropriate. There are two rules that govern how implicit CAN-HAVE-PART relationships are created. One, a class inherits all of its superclasses' CAN-HAVE-PART relationships. Two, a class gets all of its parts' subclasses as parts.

Therefore, in this Step it is determined whether there are any such CAN-HAVE-PART's and if so they are created.

2A-6.

INSERT INTO Class_m (obj1, obj2, has_array, minimum, maximum) VALUES (?, ?, ?, ?, ?)

Appropriate values are provided for "?"'s at runtime.

2A-7.

This Step tests if any CAN-HAVE-PART relationships were specified in the OSQL CREATE CLASS statement. Number of CAN-HAVE-PART relationships specified is an argument passed to the CREATE CLASS function.

2A-8.

Step 2A-8 is similar to Step 2A-6.

3-1.

There are four ways to identify a method. They are listed in order of most efficient (ie. less disk accesses) to least efficient:

a) method_id - NO inheritance possible
b) class_id, method_name
c) class_name, method_name
d) object_id, method_name

Each choice is sufficient to uniquely identify the method.

If b) c) or d) is chosen then the object_id is first determined for the method before we go on. Because methods

can be inherited, the method may not be attached to the class the current object belongs to. In this case, it is necessary to go up the class hierarchy to find it. Steps to find methods of superclasses are described in Steps 3-3 and 3-4.

3-2.

The purpose of Steps 3-2 through 3-4 is to find the object id of the method. If method id was given to the EXECUTE_METHOD function, Step 3-6 is the next Step; otherwise, Steps 3-3 and 3-4 are used.

3-3.

The following is the SQL statement that retrieves all superclasses of the current object:

```
SELECT obj2, priority
    FROM Inheritance_m
    WHERE obj1 = < current class object id >;
```

3-4.

For each superclass of the current object's class the method is located using the following SQL statement:

```
SELECT object_id, ref_class_id
    FROM Method_i
    WHERE object_name = < method name >
        AND ref_class_id = < current class's object id >;
```

3-5.

Similar to Step 3-4, except that the superclass is the Root class.

3-6.

Tests if the method id was determined by testing the return code of Step 3-5.

3-7.

Some information about the method is retrieved.

```
SELECT method_type, table_offset, module_name, stub_name
    FROM Method_i
    WHERE object_id = < method's object id >
```

table_offset will be used to determine which "real" function is associated to the method. In ODB, there is an array of the following structure:

```
typedef struct
{
    char          class_name[TABNAMELEN];
    char          method_name[NAMELEN];
    long          (*pfunc)();
} LINKED_METHOD;
```

A method is either already linked into the ODB process or resides in a dynamic link library. Linked-in methods have an entry in the above array. It can be located by using table_offset obtained from above SELECT statement to index into the array to get the function pointer.

Dynamic functions will be loaded into the process at runtime when they are referenced. After they are loaded they will have an entry in the LINKED_METHOD array too.

3-8.

Tests if the method is already linked in.

3-9.

If a method is not linked in yet find out which dynamic link library it resides in.

3-10.

Tests if the method was located successfully by testing the return code of Step 3-9.

3-11.

Use the Dynamic Link Library facility to link in the method.

3-12.

```
SELECT argument_type, argument_seq
    FROM MethodAttributes
    WHERE method_id = < method's object id >;
```

Type checking is done for each argument of the method.

3-13.

Once the method function is located (statically or dynamically) it can be executed with the arguments passed in.

3-14.

ODB communicates with client hosts through RPC. Part of the RPC mecanism is serialize/deserialize data. After executing the method, if there is any output data to return to the client ODB serializes it and gives it to the underlying RPC layer to pass on to the client.

3-15.

See Step 3-14.

4-1.

In this Step the class id of the object is determined. First, determine the name of the class the object belongs to.

```
SELECT class_name
    FROM ObjectID
    WHERE object_id = < id of the object >;
```

Then, determine the id of the class from its name,

```
SELECT object_id
    FROM Class_i
    WHERE object_name = < class_name retrieved from above >;
```

4-2.

```
SELECT attribute_id, attribute_type, flags
    FROM ObjectAttributes
    WHERE lowerc_attr_name = < attribute name >
        AND object_id = < object id of the class the
                          current object belongs to >;
```

4-3.

By testing the return code of Step 4-2 one can know if the attribute belongs to the class the object belongs to. If

the attribute does not belong to this class it may be inherited from some superclass.

4-4.

The attribute could be an intrinsic attribute defined for the object or it could be a shared attribute. Since value of shared attributes is stored on ValidValues table different SQL statements are used to retrieve it.

For shared attributes:

```
SELECT value_storage
    FROM ValidValues
WHERE attribute_id = <  Id of the attribute
                        determined in Step 4-1 >;
```

For intrinsic attributes:

```
SELECT < Attribute name >
    FROM < table name of the class the object belongs to>
    WHERE object_id = < object_id of the current object>;
```

4-5.

The attriubte's value is serialized and given to the RPC layer to return to the client process.

4-6.

The attribute being sought may be inherited from some superclass in the class hierarchy. So if the SELECT statement in Step 4-2 failed it is necessary to go up the class hierarchy and try to find the attribute on every

superclass along the path. Following is the SQL statement for getting object_id of the superclass.

```
SELECT obj2
    FROM Inheritance_m
    WHERE obj1 = < the object_id of the "current" class in
                   which we are looking for the attribute >;
```

4-7.

By testing the return code of Step 4-6 it can be determined if there is any superclass.

4-8.

This Step is similar to Step 4-2, except the class is the Root class.

4-9.

By testing the return code of Step 4-8 it can be determined if the attribute belongs to Root class.

4-10.

This Step is similar to Step 4-4, except the class is the Root class.

4-11.

This Step is similar to Step 4-2.

4-12.

By testing the result of Step 4-11 it is determined if a particular superclass has the attribute defined.

4-13.

This Step is similar to Step 4-4.

5-1.

The OSQL SELECT command string is parsed in this Step. The outcome of the parsing is a structure that contains various information about the OSQL command. This structure will be used by ODB from this point on to fulfill the request.

5-2.

In ODB, there are different ways of organizing and relating objects together. OSQL SELECT provides a way to traverse each different hierarchy of objects.

In sub-flowcharts A), B), C), D), and E), five different ways of traversing object hierarchy are described. Based on the structure created in Step 5-1 ODB knows which type of hierarchy it is dealing with and will build a SQL SELECT statement.

In remaining Steps of this flowchart ODB will take the SQL SELECT built in A), B), C), D) or E) and retrieve data from underlying SQL database.

In this Step, the SQL data dictionary is queried to allocate enough memory to hold information about every attribute to be retrieved from the database.

```
PREPARE selectit FROM $cmd;
DESCRIBE selectit INTO sqldaptr;
```

5-3.

Since there may be multiple records retrieved from the database the result of executing the SQL SELECT is put into a cursor. The cursor will then be FETCHed sequentially and stored in a buffer which will eventually be returned to the calling function. DECLARE cursor1 CURSOR FOR selectit;
OPEN cursor1;

This Step and next Step retrieve records from the cursor filled up in Step 5-3 and put them into a buffer to be returned to the client.

Each record is FETCH'ed from the cursor into a dynamic buffer allocated in Step 5-2 as follows.

FETCH cursor1 USING DESCRIPTOR sqldaptr;

Then each field of the record is moved from the dynamic buffer into another buffer which eventually is returned to the client.

5-5.

See Step 5-4.

5-6.

The buffer prepared in Steps 5-4 and 5-5 is given to the RPC layer to return to the client.

6-1.

Initializes the command buffer to nulls.

6-2.

The basic principle of translating an OSQL SELECT into SQL SELECT is to build a single SQL statement based on the semantics of the OSQL statement, then give it to the underlying SQL layer.

For SELECTing FROM CLASS, the SQL eventually built will look something like this:

```
SELECT < attribute_list_as_defined_by_OSQL_SELECT >
    FROM < Class_name_i >
    WHERE < any_appropriate_conditions >;
```

This statement is dynamically built at runtime, the exact syntax is determined by the OSQL statement.

6-3.

Move the SQL statement into the command buffer to be returned.

6-4.

The structure established in Step 6-1 will be used to determine type of SELECT this is.

6-5.

Return the built SQL statement to the calling function. Proceed to Step 6-2.

7-1.

All sub-class's id's are retrieved for the class the current object belongs to. Since there may be multiple subclasses these class id's are saved into a cursor.

```
DECLARE scbuf CURSOR FOR
    SELECT subclass
    FROM subclass
    WHERE superclass = < class object id >;

OPEN scbuf;
```

7-2.

Subclass id's in the cursor set up in Step 7-1 are retrieved one by one to get the class names for these sub-classes.

```
SELECT object_name
    INTO < some variable to hold the class name >
    FROM class_i
    WHERE object_id = < subclass id >;
```

These class names are saved in a character array for later use.

7-3.

Tests the status code of Step 7-2 to see if any subclass was found.

7-4.

If no subclass was found return to the calling function in error.

7-5.

A SELECT SQL statement is built for each of the sub-class and will be UNION'ed togather to form a single SQL statement.

```
SELECT < attribute_list_as_defined_by_OSQL_SELECT >
    FROM    < sub_class_name_i >
    WHERE   < any_appropriate_conditions >;
```

7-6.

Append the SQL SELECT statement into the command buffer and continue the loop.

8-1.

```
SELECT class_name
    FROM ObjectID
    WHERE object_id = < object id of the current object >;
```

8-2.

```
SELECT DISTINCT object_name
    FROM Class_i, Parts_m
    WHERE Parts_m.ref_class_id = Class_i.object_id
          AND
          Parts_m.obj1 = < object_id_of_current_object >;
```

This statement retrieves all part classes for the current object. The result is put in a cursor because there may be multiple rows.

8-3.

Initializes the command buffer to nulls.

8-4.

This Step FETCH'es a part class name from the cursor set up in Step 8-2.

8-5.

By testing the return code of Step 8-4 it is determined if any part class was found.

8-6.

If no part class was found, return in error.

8-7.

For each part class retrieved in Step 8-1 a SELECT statement is built as below. These SELECT's will be UNION'ed together to build the final SQL. Due to the fact that these individual SELECT's are UNION'ed only "common" attributes of part classes can be selected.

```
SELECT < attribute_list_as_specified_by_OSQL_statement >
    FROM part_class_name_i, Parts_m
    WHERE Parts_m.obj1 = < Object_id_of_the_current_object >
        AND Parts_m.obj2 = part_class_name_i.object_id;
```

8-8.

Append the SQL SELECT statement into the command buffer and continue the loop.

9-1.

```
SELECT class_name
    FROM ObjectID
    WHERE object_id = < object id of the current object >;
```

9-2.

SELECT FROM AGGREGATE is very similar to SELECT FROM PART. Following is the SQL statement that retrieves all aggregate classes of the current object.

```
SELECT DISTINCT ObjectId.class_name
    FROM ObjectId, Parts_m
    WHERE Parts_m.obj2 = < object_id_of_current_object >
        AND Parts_m.obj1 = ObjectId.object_id;
```

9-3.

Initializes the command buffer to nulls.

9-4.

This Step FETCH'es an aggregate class name from the cursor set up in Step 9-2.

9-5.

Similar to Step 9-2, a SELECT statement is built for each aggregate class of the current object. Then all these SELECT's will be UNION'ed together to make a single statement.

```
SELECT < attribute_list_as_specified_by_OSQL_statement >
    FROM < aggregate_class_name_i >, Parts_m
    WHERE Parts_m.obj2 = < object_id_of_current_object >
        AND Parts_m.obj1 = part_class_name_i.obj;
```

9-6.

Append the SQL SELECT statement into the command buffer and continue the loop.

10-1.

```
SELECT < attribute_list_as_defined_by_OSQL_SELECT >
    FROM    Link_i
    WHERE   < any_appropriate_conditions >
```

This statement is dynamically built at runtime, the exact syntax is determined by the OSQL statement.

11-1.

SELECT FROM LINKED is similar to SELECT FROM PART or SELECT FROM AGGREGATE. Following is the SQL statement that retrieves all linked classes of the current object.

```
SELECT DISTINCT ObjectId.class_name
    FROM ObjectId, Link_i
    WHERE (
           ObjectId.object_id = Link_i.obj1
           AND Link_i.obj2 = < object_id_of_current_object >
          )
          OR
          (
           ObjectId.object_id = Link_i.obj2
           AND Link_i.obj1 = < object_id_of_current_object >
          );
```

11-2.

Initializes the command buffer to nulls.

11-3.

This Step FETCH'es a linked class name from the cursor set up in Step 11-1.

11-4.

Similar to Step 9-2, a SELECT statement is built for each linked class of the current object. Then all these SELECT's will be UNION'ed together to make a single statement.

```
SELECT < attribute_list_as_defined_by_OSQL_SELECT >
    FROM part_class_name_i, Link_i
    WHERE part_class_name_i.object_id = Link_i.obj1
        AND Link_i.obj2 = < object_id_of_current_object >
UNION ALL
SELECT < attribute_list_as_defined_by_OSQL_SELECT >
    FROM part_class_name_i, Link_i
    WHERE part_class_name_i.object_id = Link_i.obj2
        AND Link_i.obj1 = < object_id_of_current_object >;
```

11-5.

Append the SQL SELECT statement into the command buffer and continue the loop.

12-1.

This Step tries to locate the new method in LINKED_METHOD array described in Step 3-7.

12-2.

If the method is not already linked in it should be a dynamic method. This Step tries to get the name of the Dynamic Link Library where the new method can be found.

12-3.

Creating a method object is similar to creating other kinds of objects in ODB. The only difference is it is a kind of "meta" object. The system keeps information about these meta objects. For example, there is a Method_i table and a Methodattributes table. ODB uses Method_i table to store method objects and Methodattributes table to store method arguments.

Since it is desired to create a method object the attributes of a method object are determined by retrieving them from the Objectattributes table.

```
SELECT  attribute_id, attribute_name, attribute_type,
inherited, table_number, flags
    FROM ObjectAttributes
    WHERE object_id = < An integer constant representing the
                        object id of Method meta class >;
```

The above SELECT statement will be PREPARE'd and a cursor will be DECLARE'd and OPEN'ed to hold all the attributes information about the Method_i table.

12-4.

Then it is desired to build a SQL statement dynamically at runtime to INSERT an entry into the Method_i table. Following is the statement:

```
INSERT INTO Method_i ( object_id, object_type, class_name,
                       time_created, created_by,
                       time_modified, modified_by, ... )
    VALUES ( < determined at runtime > );
```

As shown above, the INSERT statement will be created dynamically and eventually execute. After an entry is INSERT'ed into the Method_i table for the new method sucessfully, a new method object is created.

12-5.

After a new method is created the information about its arguments is inserted into the Methodattributes table as follows,

```
INSERT INTO MethodAttributes ( method_id, argument_type,
                               argument_seq )
    VALUES ( < id of the new method >, < argument type >,
             < argument sequence number > )
```

13-1.

To create any object the attributes of the class this object belongs to must be known so they are retrieved from the Objectattributes table.

```
SELECT  attribute_id, attribute_name, attribute_type,
        inherited, table_number, flags
  FROM ObjectAttributes
  WHERE object_id = <id of the class the object belongs to>;
```

The above SELECT statement will be PREPARE'd and a cursor will be DECLARE'd and OPEN'ed to hold all the attributes information about the class.

13-2.

Similar to 13-2 a SQL statement is built dynamically at runtime to INSERT an entry into the table for the class.

```
INSERT INTO Method_i ( object_id, object_type, class_name,
                       time_created, created_by,
                       time_modified, modified_by, ... )
    VALUES ( < determined at runtime > );
```

As shown above, the INSERT statement will be created dynamically and eventually execute.

14-1.

Again, creating an object in ODB follows the same flow of actions: a) get class description, b) insert an entry into the table for the class. Creating a link object is no

exception. Attribute information comes from Objectattributes table and link objects are stored on Link_i table.

```
SELECT  attribute_id, attribute_name, attribute_type,
        inherited, table_number, flags
    FROM ObjectAttributes
    WHERE object_id = <  An integer constant representing
                         the object id of Link meta class >;
```

The above SELECT statement will be PREPARE'd and a cursor will be DECLARE'd and OPEN'ed to hold all the attributes information about the Link_i table.

14-2.

Then it is desired to build a SQL statement dynamically at runtime to INSERT an entry into the Link_i table.

```
INSERT INTO Link_i ( object_id, object_type, class_name,
                     time_created, created_by,
                     time_modified, modified_by, ... )
    VALUES ( < determined at runtime > );
```

As shown above, the INSERT statement will be created dynamically and eventually execute.

15-1.

Before an object is added to another object as a part it must be determined that there is a possiblepart relationship existing between their classes.

```
SELECT has_array, minimum, maximum
    FROM Class_m
    WHERE obj1 = < class id of the owned object >
    AND obj2 = < class id of the owned object >;
```

15-2.

Next, it is desired to update part order. Parts order is kept in ODB so that part can be retrieved in a controlled manner. To update part order the maximum part order is obtained for this "type" of parts.

```
SELECT MAX(part_order)
    FROM Parts_m
    WHERE obj1 = < object_id of the owner object >
        AND ref_class_id = < class id of the class the
                             owned object belongs to >;
```

Then the ordering of parts, is updated

```
UPDATE Parts_m
    SET part_order = part_order + 1
    WHERE part_order >= < position of the new part >
        AND obj1 = < object id of owner >
        AND obj2 != < object id of part >
        AND ref_class_id = < class id of part object >;
```

15-3.

Finally, an entry is inserted into the Parts_m table.

```
INSERT INTO Parts_m ( obj1, obj2, has_array,
                      part_order, ref_class_id )
    VALUES ( < object id of the owner object >,
             < object id of the part object >,
             < is the part an intrinsic object ? >,
             < new part's position >,
             < class id of the new part > );
```

16-1.

Before an object is removed from another object as a part it must be determined that there is a possiblepart relationship existing between their classes.

```
SELECT has_array
    FROM Class_m
    WHERE obj1 = < class id of the owned object >
        AND obj2 = < class id of the owned object >;
```

16-2.

If the verification succeeded in 16-1, the part relationship can be deleted from Parts_m.

```
DELETE FROM Parts_m
    WHERE obj1 = < object id of the owner object >
    AND obj2 = < object id of the owned object >;
```

16-3.

Finally, the ordering of parts is updated.

```
SELECT MAX(part_order)
    FROM Parts_m
    WHERE obj1 = < object id of the owner object >
        AND ref_class_id = < class id of the owned object >;

UPDATE Parts_m
    SET part_order = part_order - 1
    WHERE part_order > < position of the old part object >
        AND obj1 = < object id of the owner object >
        AND ref_class_id = < class id of the part object >;
```

While the above description of the sets of object level core program commands and the technique of translating such commands in relational data base SQL call routines adequately describes the invention in a general way, in better understanding the use of the invention, it is further helpful to provide below a more specific example thereof in a particular application context. An appropriate example selected for such purpose is that of a banking system which, although reasonably simplified, adequately illustrates the techniques involved for understanding the invention.

The system is illustrated in FIG. 17 and includes a "root" class (sometimes referred to as a "master" class) which effectively represents a universal class which provides a base class from which specific classes of a system can be built or derived. Thus, in FIG. 17, the root class provides a base from which a Bank_Accounts class and a Customers class can be built.

Likewise, Bank_Accounts provides a base from which the Checking and Savings classes can build. Finally, the Savings class provides the base for the subclass, Super_Savings. Because each class is a type of its parent class, it inherits certain information and procedures from classes above it. The IS_TYPE_OF relationship is shown by the arrows pointing downwards to illustrate the concept of inheritance.

From such illustration it can be seen how classifying customers and accounts provides coherence for the system. Adding inheritance through a class hierarchy provides advantages to object data base systems not available when using the traditional file structures. Inheritance permits one to define a characteristic for a class and then to use it automatically for all of the subclasses of that class.

In addition to inheritance, a second important feature supported by ODB systems lies in the ability of a class to be part of another class. FIG. 18 shows this for the class Customers.

In FIG. 18, the line connecting the classes Customers and Bank_Accounts indicates that Customers can own Bank_Accounts, i.e., the latter can be a part of the former, or put in another way the class Customers has a CAN-HAVE-PARTS relationship to Bank_Accounts. Because Bank_Accounts has the sub-classes Checking, Savings, and Super_Savings, Customers can own those accounts as well. Thus, a CAN-HAVE-PARTS relationship lets one define or constrain a class as being able to consist of other classes. Defining or constraining the class Customers permits individual specified customers (instances) to have none, one, or more than one of each kind of account.

Information about a class is called the attributes of a class. Procedures associated with a class are called methods. FIG. 19 shows how information about a class, i.e., attributes, relates to classes.

FIG. 19 shows how an ODB stores attributes, or information, about a class. Attributes can be text strings, integers, buffers, or files. For example, text strings are useful for identifying an object by its name; integers are used for providing coded identifications of classes for representing numerical facts about an object such as account balances; buffers are used for storing data when accessing the ODB; and files can store large amounts of information related to an object.

In FIG. 19, attributes that apply to a class appear next to the class where the attribute is first defined. Subclasses inherit those attributes and add additional attributes. For example, the attributes Obj_Id, Obj_Name, and Date_Time_Created, defined once for the Root class, apply to all subclasses. Attributes can be shared by different classes. For example, all instances of Savings account share the attribute Int_Rate. The sharing of attributes saves considerable space in terms of data storage. An attribute created for a subordinate class, such as Cust_Name for Bank_Accounts, is inherited by all of Bank Accounts' subclasses. While attributes can be added for a class, attributes of a class can't be changed.

Methods, or procedures, are another kind of entity with which inheritance can be used. The ability to store methods makes an ODB different from a relational data base which stores only passive data. FIG. 20 illustrates for the banking example the concept of methods.

Unlike an attribute, a method is a class object and, like attributes, methods are defined for a class, are stored with it, and are inherited. For example, Deposit and Withdraw methods are defined for Bank_Accounts and apply as well to Checking, Savings, and Super_Savings, i.e., inherited by them. It should be noted however, that an inherited method can be modified if necessary. For example, a Withdraw method associated with

SuperSavings class accounts is different from a Withdraw method associated with other kinds of accounts. Likewise, a Credit_ Interest method for Savings can differ from a Credit_Interest method for Super_Savings because of a minimum balance and higher rate of interest for the Super_Savings accounts.

The storage of methods and how such methods are executed are two major differences between programming used for ODB operation and used for conventional data bases. In conventional programming, it is customary to use libraries of procedures. Typically, however, binding precedes execution by some long period of time. Because procedures are stored in programs, a change in a library procedure means that all programs that use the procedure need to be recompiled and relinked. In an ODB, on the other hand, methods and classes are stored in tight conjunction with each other. There is never any question about which procedures need to be reworded. In the ODB programming environment, only one copy of a method is "live" and in the ODB environment the way a program is executed is different; a program activates a class method by sending a message to the object. Thus, data and procedures are not just associated in the ODB, they are intimately connected.

FIG. 21 illustrates the concept of instances where an instance is a member of its defining class.

Just as there is a class hierarchy, there is also an instance hierarchy. FIG. 22 shows the relationship of instances to other instances wherein Jane-Customer has two accounts, and also holds some Treasury bills. The lines connecting Jane_ Customer and her checking and savings accounts show the HAS-PARTS relationship, determined by the CAN-HAVE-PARTS relationship defined for classes as discussed above. The dotted line between Jane and TBill_03 shows another possible feature of an instance hierarchy, the HAS-LINK relationship. Here, a Treasury bill is not like a savings or checking account in that the bank merely sells it to the customer. The bank doesn't need to keep track of interest, maturity date, or in any other way manage the Treasury bill on behalf of the customer, but does like to know which customers have bought Treasury bills in the past.

The foregoing FIGS. 17-22 point out the difference between classes and instances. Classes give shape in generalities, defining all of the data structures that must be filled in for each customer, as well as registering the methods appropriate for dealing with each kind of action. An instance shows the specific facts for each customer.

Having described the exemplary banking system above, a data base can then be created using such example, the following discussion describing what occurs within the system when such a

data base is created in the environment of the invention. In such description, an example is given of the creation of meta tables using appropriate SQL statements which are familiar to those in the art. An example of how a banking data base is created using object SQL statements is then provided, followed by an example of a number of object SQL statements which demonstrate how objects can be manipulated.

Creation Of An Underlying SQL Database And Meta Tables

To create a banking database, for example, it is necessary to first create "meta tables" which store "bootstrap" information for the system to start functioning. Examples of these meta tables are Class_i, Method_i, ObjectId, ObjectAttributes, and so on. A complete list of meta tables is set forth in Appendix A for the banking example being discussed.

Meta tables are created by a utility program which makes SQL requests directly. Following is a list of SQL statements that accomplish such task.

First of all, a SQL database is created.

```
CREATE DATABASE $dbname WITH BUFFERED LOG;
```

Then, all the meta tables are created. For example, in order to create a ObjectId meta table the following SQL statement is used;

```
CREATE TABLE ObjectId
     (object_id          LONG          NOT NULL,
      object_type        SMALLINT      NOT NULL,
      real_object_id     INTEGER,
      class_name         CHAR(17)      NOT NULL,
      object_state       INTEGER,
      object_uid         INTEGER,
      use_count          SMALLINT,
      object_rev         CHAR(16)      NOT NULL,
      next_child_rev     CHAR(16)      NOT NULL);
```

Other meta tables are created in substantially the same way, the column definitions being different among different tables. Creation of other meta tables would accordingly be understood by those in the art from the above examples.

User-defined tables are created to hold user-defined objects. Each object will be represented by a "row" in the table for its class. The table name is made up of the class name appended with "_i". Attributes of a class are repeated in each subclass's table for efficiency reasons. User-defined

tables for the banking examples under discussion were listed above and set forth in Appendix B.

Object SQL Statements To Create A Banking Database

To create any object SQL database the first step is create the "classes" therein and to relate them to each other. For example, the following OSQL statements are used to create classes for a banking database.

```
CREATE CLASS bank_accounts
     ATTRIBUTES (balance LONG);

CREATE CLASS customers
     ATTRIBUTES (cust_address CHAR(100),
                 cust_phone CHAR(20),
                 irs_num CHAR(12))
     PARTS (bank_accounts MIN 0 MAX 5);

CREATE CLASS checking
     SUPERCLASSES (bank_accounts)
     ATTRIBUTES (charge_class CHAR(2),
                 overdraft_limit LONG);
```

```
CREATE CLASS savings
     SUPERCLASS (bank_accounts)
     ATTRIBUTES (int_rate SHORT = 8 SHARED,
                 credit_date CHAR(2));

CREATE CLASS super_savings
     SUPERCLASSES (savings)
     ATTRIBUTES (int_rate SHORT = 9 SHARED,
                 minimum_balance LONG);

CREATE CLASS TBILLS
     ATTRIBUTES (amount LONG,
                 issue_date CHAR(20),
                 due_date CHAR(20),
                 interest_rate SHORT);
```

After creating classes objects using these classes can be created as "templates". The following is a list of OSQL statements that create the instance objects of the above discussed banking example, namely, Jane_customer for Customers class, Checking_Acct_01 for Checking class, Savings_Acct_02 for Savings class and Tbills_03 for TBILLS class.

```
CREATE OBJECT
     OF CLASS Customers (object_name "Jane_customer",
                         cust_address "4 Elm St.,
                         Westboro, MA  01580",
                         cust_phone "508-870-6000",
                         irs_num "022-45-7700");

CREATE OBJECT
     OF CLASS Checking (object_name "checking_acct_01",
                        balance 10000

CREATE OBJECT
     OF CLASS Savings (object_name "savings_acct_02",
                       balance 10000,
                       credit_date "15");

CRLEATE OBJECT
     OF CLASS TBILLS (object_name "Tbills_01",
                      amount 10000,
                      issue_date "1-Jan-90",
                      due_date "31-Dec-90");
```

These objects can be related to each other to show, for example, that customer Jane owns two bank account objects, one for saving and one for checking. To do so, checking_acct_01 and savings_acct_02 are "added" as "parts" to Jane_customer.

```
ADD   object_id of checking_acct_01
     TO  object_id of Jane_customer  ;

ADD   object_id of savings_acct_02
     TO  object_id of Jane_customer  ;
```

Finally, Jane_customer can be related to Tbills_01 to show that customer Jane purchased Tbills_01 in the past. To do that, a "Link" is created between Jane_customer and Tbills_01.

```
CREATE LINK
     BETWEEN   object_id of Jane_customer
         AND   object_id of Tbills_01  ;
```

Examples of OSQL Statements That Manioulate
Objects In the Banking Database

After the banking database is created the objects therein can be manipulated. Objects can be modified, deleted, or information about objects can be retrieved from the database, for example, some OSQL statements that retrieve information from the banking database and activate "code" that produces useful information, are discussed below.

To retrieve names of all the accounts Janes owns:

```
SELECT object_name
     FROM PART OF  object_id of Jane_customer  ;
```

To retrieve names of all Savings accounts Jane owns:

```
SELECT object_name
     FROM PART Savings Of  object_id of Jane_customer  ;
```

To retrieve Customer's name who owns Savings Account "savings_acct_02";

```
SELECT object_name
     FROM AGGREGATE Customers OF  object_id of
          savings_acct_02  ;
```

To see the balance of Jane's Checking Account:

```
SELECT balance
     FROM PART Checking OF  object_id of Jane_customer  ;
```

To generate a monthly printout for Jane:

```
EXECUTE METHOD Print_Acct_Balances
     ON   object_id of Jane_Customer  ;
```

The above description provides information concerning the invention both in a general way and by way of a specific example (of a banking system) which permit those of skill in the art to practice the invention. While the particular embodiment of the invention disclosed above describes is preferred, modifications thereto will occur to those in the art within the spirit and scope of the invention. Hence, the invention is not to be construed as limited thereto except as defined by the appended claims.

APPENDIX A

Tables For Meta Information

This Appendix gives a detailed description of the meta tables listed in the specification. It should be noted that there is a Root Class (sometimes referred to as a Master Class) meta table which is the highest class and every user-defined class must inherit from such Root Class. In a particular application, if a meta table does not have any columns listed it will still have those Root Class columns defined for them.

[********** Root **********]

Column name	Type	Nulls
object_id	integer	no
| unique id of an object |		
object_name	char(32)	yes
| name of an object, does not have to be unique |		
object_type	smallint	no
| type of object is one of class, instance, method, | link, alerter or timedalerter |		
class_name	char(16)	no
| name of the class the object belongs to |		
time_created	integer	no
| the time object was created |		
created_by	char(64)	yes
| name of the person who created the object |		
time_modified	integer	no
| the time object was modified |		
modified_by	char(64)	yes
| name of the person who modified the object |		

[********** Class **********]

Column name	Type	Nulls

[********** Class_m **********]

Column name	Type	Nulls
obj1	integer	no
	object id of the owner class	
obj2	integer	no
	object id of the owned class	
has_array	smallint	no
	is the part object an intrinsic part of the owner object	
minimum	smallint	no
	minimum number of owned parts of the owner object | must have	
maximum	smallint	no
	maximum number of owned parts of the owner object | can have	

[********** Inheritance_m **********]

Column name	Type	Nulls
obj1	integer	no
	object id of the super class	
obj2	integer	no
	object id of the sub class	

[********** Link_i **********]

Column name	Type	Nulls
obj1	integer	no
	object id of one object	
obj2	integer	no
	object id of another object	
description	char(200)	yes
	any information about ths link	

[********** Method_i **********]

Column name	Type	Nulls
ref_class_id	integer	no
	object id of the class the method is attached to	

```
method_type          smallint            no
    | a linked_in method or a dynamic method |
table_offset         smallint            yes
    | for a linked in method, where in the internal table |
    | is the method |


[********** MethodAttributes **********]

Column name          Type                Nulls
-----------------    --------            -----

method_id            integer             no
    | object id of the method |
argument_type        integer             no
    | data type of an argument |
argument_seq         smallint            no
    | order of an argument of a method |


[********** ObjectAttributes **********]

Column name          Type                Nulls
-----------------    --------            -----

object_id            integer             no
    | object id of the class |
attribute_id         integer             no
    | unique id of an attribute |
attribute_name       char(32)            no
    | name of the attribute |
lowerc_attr_name     char(32)            no
    | lower case attribute name |
attribute_type       char(8)             no
    | an attribute is one of the following types: char, long,
      | short, file or buffer. |
inherited            integer             yes
    | object id of the superclass from which the attribute |
    | is inherited |
flags                smallint            no
    | indicates if the attribute is modifiable, |
    | shared, inherited, default, limited, public, private, |
    | or required. Each bit is a flag. |


[********** ObjectId **********]

Column name          Type                Nulls
-----------------    --------            -----

object_id            integer             no
    | unique id of the object |
```

```
object_type          smallint            no
    | type of object is one of class, instance, method, |
    | link, alerter, timedalerter |
class_name           char(16)            no
    | name of the class the object belongs to |
object_state         integer             yes
    | objects can be in different states.  ODB keeps |
    | objects in the following states: enabled, deleted,|
    | checked_out, frozen, shadow, delete_locked, |
    | retained_locked, unavailable, available |
object_uid           integer             yes
    | the user id of the object. |


[********** Parts_m **********]


Column name          Type                Nulls
-----------------    --------            -----

obj1                 integer             no
    | object id of the owner object |
obj2                 integer             no
    | object id of the owned object |
has_array            smallint            yes
    | is the part object an intrinsic part of the owner object |
part_order           smallint            no
    | order of the part object |
ref_class_id         integer             no
    | class id of the part object |
```

APPENDIX B

Tables For User-Defined Classes

This Section gives a detailed description of the User_Defined tables listed in the specification for the particular banking example set forth therein.

[********** Customers_i **********]

Column name	Type	Nulls
cust_address	char(100)	yes
cust_phone	char(20)	yes
irs_num	char(12)	yes

[********** Bank_Accounts_i **********]

Column name	Type	Nulls
balance	integer	yes

[********** Checking_i **********]

Column name	Type	Nulls
balance	integer	yes
charge_class	char(2)	yes
overdraft_limit	integer	yes

[********** Savings_i **********]

Column name	Type	Nulls
balance	integer	yes
credit_date	char(2)	yes

[********** Super_Savings_i **********]

Column name	Type	Nulls
balance	integer	yes
credit_date	char(2)	yes
minimum_balance	integer	yes

[********** Tbills_i **********]

Column name	Type	Nulls
amount	integer	yes
issue_date	char(20)	yes
due_date	char(2)	yes

CLAIMS

1. A method of creating objects in a relational data base in accordance with a data base schema, which data base responds to commands and command call routines at a relational data base level, said method comprising the steps of

developing a data base schema generation program using abstract data base schema generation commands at a host program language level;

preprocessing said abstract data base schema generation commands to encode said commands into expanded command call routines at said host program language level;

automatically converting said expanded command call routines into data base schema generation call routines at said relational data base language level for use in creating objects in said relational data base, said objects containing information in accordance with said data base schema.

2. A method of manipulating objects created in a relational data base in accordance with claim 1 and further including the steps of

developing an applications programs for manipulating said objects using abstract applications commands at said host program language level;

preprocessing said abstract applications commands to encode said commands into expanded applications command call routines at said host program language level;

said converting step automatically converting said expanded applications command call routines into applications command call routines at said relational data base language level for use in manipulating the objects in said relational data base in accordance with said applications program.

3. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object-oriented create class command call routine into a create class command call routine at said relational data base language level.

4. A method in accordance with claim 3 wherein said create class command call routine converting step includes the steps of

(a) determining whether the class to be created already exists in said relational data base;

(b) if said class does not exist, inserting a new entry for identifying said class in a class table of said relational data base;

(c) determining any attributes and possible parts to be inherited by said class from another class existing in said relational data base;

(d) determining any new attributes and possible parts for said class;

(e) entering said inherited and said new attributes in an instance class table for said class in said relational data base; and

(f) entering said inherited and said new possible parts in a possible parts table for said class in said relational data base.

5. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented execute method command call routine into an execute method command call routine at said relational data base level for manipulating information in said relational data base.

6. A method in accordance with claim 5 wherein said converting step includes

(a) determining if said execute method call routine already exists in said relational data base;

(b) if said execute method call routine does not exist, determining if said execute method call routine is inherited from an existing class in said relational data base;

(c) if said execute method call routine already exists or is inherited, obtain the function code of said execute method call routine if said function code is already linked to the object data base process;

(d) if said function code is not so linked, obtain the function code from a specified library thereof;

(e) when the function code is obtained, execute the method defined by said function code.

7. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented get attribute command call routine into a get attribute command call routine at said relational data base language level.

8. A method in accordance with claim 7 wherein said converting step includes the steps of

(a) determining the class of the object having said attribute;

(b) determining if a class attribute table of said object has said attribute;

(c) if said class attribute table has said attribute, retrieving the value of said attribute from an instance table of said object.

9. A method in accordance with claim 8 and further including the steps of

(d) if said class attribute table does not have said attribute, retrieving said attribute from a class attribute table of another class which is a superclass of said class; and

(e) if a class attribute of a superclass does not have said attribute, retrieving said attribute from a class attribute table of a master class.

10. A method in accordance with claim 1 wherein said converting step includes selecting an attribute from a plurality of attribute tables.

11. A method in accordance with claim 10 wherein selecting an attribute includes

determining from which of a plurality of attribute tables the attribute is to be selected, said attribute tables being respectively for a superclass, a class, a part, an aggregate, a link or a linked;

constructing a relational data base select attribute table command call routine for selecting one of said attribute tables;

determining the amount of memory required to store the attributes retrieved from the selected attribute table by said select attribute table command call routine.

12. A method in accordance with claim 11 wherein said attribute table command call routine is a class select command call routine for retrieving attributes from an attribute table for a specified class and all subclasses of said specified class.

13. A method in accordance with claim 11 wherein said attribute table command call routine is a part select command call routine for retrieving attributes from an attribute table for the part objects of an identified object.

14. A method in accordance with claim 13 wherein said part select command constructing step includes

(a) determining the class of said identified object;

(b) determining the classes of objects that said identified object can have as parts thereof;

(c) determining from the classes determined in step (b) if a possible part class exists for said classes;

(d) retrieving the attributes from attribute tables for each of said possible parts classes.

15. A method in accordance with claim 11 wherein said attribute table command call routine is an aggregate select command call routine for retrieving attributes from aggregate objects of an identified object.

16. A method in accordance with claim 15 wherein said aggregate select command call routine includes

(a) determining the class of said identified object;

(b) determining the aggregate classes for said identified object;

(c) retrieving attributes from the attribute tables for each of said aggregate classes.

17. A method in accordance with claim 11 wherein said attribute table command call routine is a link select command call routine for retrieving attributes from a link attributes table.

18. A method in accordance with claim 11 wherein said attribute table command call routine is a linked select command call routine for retrieving attributes from other objects linked to said identified object.

19. A method in accordance with claim 18 wherein said linked select command call routine includes

(a) determining the classes of objects linked to said identified object;

(b) retrieving the attributes of the objects linked to said identified object from attribute tables of said linked objects.

20. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented create method command call routine into a create method call routine at said relational data base level.

21. A method in accordance with claim 20 wherein said create method command call routine converting step includes

(a) determining if said method is already linked in to said data base;

(b) if said method is so linked in, obtaining a description of a method meta class for said linked method;

(c) inserting method entry and method argument information into said method meta class.

22. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented create instance command call routine into a create instance command call routine at said relational data base level.

23. A method in accordance with claim 22 wherein said create instance command call routine includes

(a) obtaining a description of a new instance's class;

(b) inserting said instance in said new instance class's instance table.

24. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented link object command call routine into a link object command call routine at said relational data base language level.

25. A method in accordance with claim 24 wherein said link object command call routine includes

(a) obtaining a description of the class of said link object;

(b) inserting the attributes of said class into the attributes of said link object.

26. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented add parts command call routine into an add parts command call routine at said relational data base language level for adding an object as a part of another object.

27. A method in accordance with claim 26 wherein said add parts command call routine includes

(a) determining if a possible parts relationship between said objects has been defined;

(b) if such a possible parts relationship has been defined, updating the ordering of the parts in said possible relationship;

(c) inserting said object to be added as an entry to the parts meta table of said other object.

28. A method in accordance with claim 1 wherein said converting step includes

automatically converting an object oriented remove part command call routine into a remove part command call routine at said relational data base language level for removing an object as a part of another object.

29. A method in accordance with claim 28 where said remove part command call routine includes

(a) determining if a possible parts relationship has been defined between said objects;

(b) if such a possible parts relationship has been defined, deleting said object to be removed from the parts meta table of said other object;

(c) updating of the ordering of the parts in said possible parts relationship.

30. Apparatus for creating objects in a relational data base in accordance with a data base schema, which data base responds to commands and command call routines at a relational data base level, said apparatus comprising:

means for developing a data base schema generation program using abstract data base schema generation commands at a host program language level;

means for preprocessing said abstract data base schema generation commands to encode said commands into expanded command call routines at said host program language level; and

means for automatically converting said expanded command call routines into data base schema generation call routines at said relational data base language level for use in creating objects in said relational data base, said objects containing information in accordance with said data base schema.

31. A method of creating objects in a relational data base, substantially as herein described with reference to the drawing.

32. Apparatus for creating objects in a relational data base, substantially as herein described with reference to the drawing.

Patents Act 1977
Examiner's report to the Comptroller under Section 17 (The Search Report)

Application number: 9117682.6

Relevant Technical fields

(i) UK Cl (Edition K) G4A (AD)

(ii) Int CL (Edition 5) G06F

Search Examiner: L A MIDDLETON

Databases (see over)

(i) UK Patent Office

(ii) ONLINE DATABASES: WPI, INSPEC

Date of Search: 27 MAY 1992

Documents considered relevant following a search in respect of claims 1-32

Category (see over)	Identity of document and relevant passages	Relevant to claim(s)
X	WO 90/04829 A2 (EASTMAN KODAK) whole document	1,30

Category	Identity of document and relevant passages	Relevant to claim(s)

Categories of documents

X: Document indicating lack of novelty or of inventive step.

Y: Document indicating lack of inventive step if combined with one or more other documents of the same category.

A: Document indicating technological background and/or state of the art.

P: Document published on or after the declared priority date but before the filing date of the present application.

E: Patent document published on or after, but with priority date earlier than, the filing date of the present application.

&: Member of the same patent family, corresponding document.

Databases: The UK Patent Office database comprises classified collections of GB, EP, WO and US patent specifications as outlined periodically in the Official Journal (Patents). The on-line databases considered for search are also listed periodically in the Official Journal (Patents).

Published 1992 at The Patent Office, Concept House, Cardiff Road, Newport, Gwent NP9 1RH. Further copies may be obtained from Sales Branch, Unit 6, Nine Mile Point, Cwmfelinfach, Cross Keys, Newport, NP1 7HZ. Printed by Multiplex techniques ltd, St Mary Cray, Kent.

BNSDOCID: <GB___2253500A__I_>